UNIVERSIDAD COMPLUTENSE DE MADRID
Facultad de Informática

SISTEMAS INFORMÁTICOS **08-09**

XLOP

XML Language-Oriented Processing

Entorno para el Procesamiento de Documentos XML mediante Gramáticas de Atributos

Autores:

ALBERTO MARTÍNEZ AVILÉS
BRYAN TEMPRADO BATTAD

Director:

JOSÉ LUIS SIERRA RODRÍGUEZ

PROYECTO DE SISTEMAS INFORMÁTICOS

2008 / 2009

XLOP

(XML Language-Oriented Processing)

Autores:
ALBERTO MARTÍNEZ AVILÉS
BRYAN TEMPRADO BATTAD

Director:
JOSÉ LUIS SIERRA RODRÍGUEZ

FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Tras muchos quebraderos de cabeza, innumerables horas de trabajo y el esfuerzo constante durante más de un año, finalmente podemos presentar el entorno XLOP.

Queremos agradecer y hacer una mención especial tanto a José Luis como a Antonio por proporcionarnos toda la ayuda necesaria y por su entera dedicación al proyecto, ellos son quienes realmente han hecho posible que este proyecto salga adelante.

-Gracias-

Se autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria como el código, la documentación y/o el prototipo desarrollado.

Bryan Jesús Temprado Battad

Alberto Martínez Avilés

Resumen

En este trabajo de Sistemas Informáticos se ha desarrollado un entorno para el procesamiento de documentos XML mediante gramáticas de atributos denominado XLOP (XML *Language-Oriented Processing*). XLOP incluye un lenguaje de especificación que permite describir aplicaciones de procesamiento XML como gramáticas de atributos, cuyas funciones semánticas son proporcionadas mediante métodos de clases Java. El entorno incluye un generador que traduce las gramáticas de atributos en implementaciones expresadas en el lenguaje de CUP (una herramienta Java para la construcción de analizadores/traductores ascendentes). XLOP soporta la evaluación on-line de los atributos (es decir, simultáneamente al procesamiento de los documentos). Así mismo, el entorno permite optimizar las implementaciones CUP mediante el cálculo de *marcadores* (nuevos no terminales definidos mediante producciones vacías). Dichos marcadores permiten albergar atributos heredados, y sus producciones *disparar* la evaluación de ecuaciones semánticas. Así mismo, bajo ciertas circunstancias, XLOP optimiza la propagación de atributos hereadados a través de cadenas generadas por recursión a izquierda, permitiendo referir directamente el valor al comienzo de la cadena. En muchos casos, esto permite procesar documentos con una cantidad de memoria que no depende de la anchura de los mismos.

A fin de probar la potencialidad de XLOP para el desarrollo de aplicaciones XML, en este trabajo se ha desarrollado mediante XLOP una aplicación no trivial en el dominio de e-Learning. La aplicación, que se denomina <e-Tutor>, permite generar tutoriales interactivos a partir de su descripción como documentos XML.

Palabras clave: XML, Gramática de Atributos, Desarrollo Dirigido por Lenguajes, Procesador de Lenguaje, Análisis LALR, Herramienta de Construcción de Procesadores de Lenguaje, JavaCC, CUP, e-Learning, Sistema Tutor

Abstract

In this work we have developed an environment for processing XML documents with attribute grammars. This environment is called XLOP (XML *Language-Oriented Processing*). XLOP provides a specification language that makes it posible to describe XML processing applications with attribute grammars. The semantic functions used in these grammars are supplied as methods in Java classes. The environment provides a generator for translating attribute grammars to CUP-based implementations (CUP is a Java tool for building bottom-up parsers/translators). XLOP gives support to an on-line attribute evaluation model (i.e., attribute evaluation is interleaved with document parsing). Also, the environment allows the optimization of the CUP implementations by computing *markers* (new non-terminals that are defined using empty syntax rules). These markers are useful for containing inherited attributes. Also, their syntax rules can be used for firing the evaluation of semantic equations. In addition, under certain reasonable assumptions, XLOP optimizes the propagation of inherited attributes through chains generated by left-recursive rules, enabling the direct referencing to the value placed at the beginning of the chain. In many cases, it makes it possible to process documents with a space that does not depend on the document width.
In order to test the feasibility of XLOP in the development of XML applications, in this work we have developed a non-trivial application in the e-Learning domain using XLOP. The application, which is called <e-Tutor>, supports the generation of interactive tutorials described as XML documents.

Keywords: XML, Attribute Grammar, Language-Driven Development, Language Processor, LALR Parsing, Compiler-Construction Tool, JavaCC, CUP, e-Learning, Tutoring System

Índice

Capítulo 1
Prólogo

1.1 Introducción

El problema abordado en este trabajo de Sistemas Informáticos es el del *procesamiento* de documentos XML. XML (eXtensible Markup Language) [Bray et al. 2008], especificación propuesta a finales de los 90 por el Word Wide Web Consortium (W3C) como evolución del estándar SGML (Standard Generalized Markup Language) [Goldfarb 1991], es un estándar *de hecho* utilizado para la representación de *documentos electrónicos*. La importancia del lenguaje radica en que la información intercambiada entre los distintos componentes de un sistema informático puede entenderse en muchas ocasiones como *documentos electrónicos*. Debido a este hecho, XML es una tecnología esencial en cualquier escenario moderno de desarrollo de *software* o de gestión de la información.

XML es un lenguaje informático que norma cómo añadir a los contenidos de un documento electrónico la *metainformación* necesaria para explicitar su estructura (p.ej., su *título*, sus *autores*, sus *capítulos*, cada *sección* en cada capítulo, etc.). Dicha metainformación consiste en un conjunto de *marcas* o *etiquetas* debidamente anidadas, que describen la estructura del documento y que, por tanto, facilitan el posterior (o posteriores) procesamiento(s) del mismo. Además de introducir un criterio estándar para marcar documentos, XML también establece mecanismos que permiten restringir los posibles usos del marcado (p.ej., en el interior de la lista de autores no puede aparecer un capítulo). Estos mecanismos se rigen por un modelo *lingüístico*: para acotar los posibles usos del marcado debe especificarse una *gramática formal*. XML incluye un sub-lenguaje para describir tales gramáticas (el sub-lenguaje de las DTDs –*Document Type Definitions*). Así mismo, la comunidad XML ha propuesto también otras muchas alternativas para describir tales *gramáticas documentales*, que extienden o complementan las capacidades expresivas de las DTDs [Murata et al. 2005].

XML, sin embargo, *no* norma cómo deben procesarse los documentos en una determinada aplicación informática. De hecho, esta separación entre estructura y procesamiento es uno de los principios básicos de XML: el marcado en XML es *descriptivo*, orientado a plasmar la estructura lógica de los documentos, pero no las posibles formas de procesar dichos documentos [Coombs et al. 1987]. El procesamiento de los documentos transciende el propósito de XML, y debe llevarse a cabo utilizando otros medios. Este hecho desemboca, por tanto, en el problema en el que se enmarca este trabajo: *cómo llevar a cabo el procesamiento de los documentos* XML.

Desde la aparición de XML se han propuesto también múltiples tecnologías para abordar el problema del procesamiento de los documentos XML. Dichas tecnologías pueden clasificarse, inicialmente, en *tecnologías específicas* y *tecnologías de propósito general*:

- Las tecnologías específicas se centran en un tipo específico de tareas (p.ej., *consulta* de la información contenida en el documento, o *transformación* del documento a otro tipo de formato). De esta forma, cada una de estas tecnologías es aplicable a dicho tipo específico de tareas (p.ej., XSLT es un lenguaje que permite especificar transformaciones de documentos [Kay 2007].

- Las tecnologías de propósito general son aplicables a cualquier tarea de procesamiento de documentos XML. Estas tecnologías se presentan, normalmente, como un conjunto de herramientas o como un marco de aplicación embebidos en un lenguaje de programación de propósito general (p.ej., Java, PHP o C#). El núcleo de una tecnología de este tipo es un analizador o parser XML: un artefacto software que lee documentos XML, comprueba el correcto uso del marcado, y ofrece una interfaz adecuada a los elementos de información contenidos en los documentos [Lam et al. 2008]. Utilizando, entonces, el lenguaje de programación genérico en el que se embebe la tecnología, se programa el procesamiento deseado: el resultado de esta actividad de programación es la lógica específica de la aplicación. Dicha lógica específica accede a la información de los documentos a través de la interfaz ofrecida por el parser, y lleva a cabo el procesamiento requerido.

Independientemente de su naturaleza, todas las tecnologías descritas comparten una característica común: el entender los documentos XML como *estructuras de datos*. Esta visión orientada a los datos contrasta, sin embargo, con la concepción lingüística original de las aplicaciones XML. Efectivamente, tal y como se ha indicado anteriormente, formular una aplicación XML es equivalente a proponer un lenguaje de marcado específico para un determinado tipo de documentos, lenguaje que viene definido mediante un modelo lingüístico formal: la gramática documental. Este hecho conduce, por tanto, a una reflexión evidente: si el objeto de procesamiento en una aplicación XML es un lenguaje, ¿*por qué no entender dicha aplicación como un procesador de lenguaje*, *y de esta forma aprovechar el enorme arsenal de conocimiento, métodos, técnicas y herramientas disponible en un dominio tan maduro como es el de la construcción de procesadores de lenguaje* (véase, por ejemplo, [Aho et al. 2007]). En este trabajo de Sistemas Informáticos se explora esta posibilidad.

1.2 Objetivos del proyecto

El principal objetivo de este trabajo de Sistemas Informáticos es la construcción de un entorno de desarrollo que permita *entender* la construcción de aplicaciones XML como la construcción de procesadores de lenguaje. Más concretamente, el entorno deberá permitir:

- Especificar cada aplicación de procesamiento XML como un procesador para el lenguaje de marcado definido por su gramática documental.

- Generar automáticamente el procesador a partir de dicha especificación.

El entorno se denominará XLOP (XML Language-Oriented Processing), y ofrecerá un lenguaje de especificación basado en *gramáticas de atributos* [Knuth 1968; Paaki 1995]. De esta forma, utilizando XLOP será posible:

- Especificar las aplicaciones de procesamiento XML como gramáticas de atributos.

- Generar automáticamente implementaciones Java de dichas aplicaciones a partir de dichas especificaciones.

Como objetivo adicional, este trabajo persigue también mostrar el potencial y el uso de XLOP en el desarrollo de una aplicación no trivial en el dominio de e-Learning. Para ello, se ha elegido refactorizar y extender <e-Tutor>, una aplicación para la generación de *sistemas tutores* [Sleeman&Brown, 1986] a partir de documentos XML.

Por último, en este trabajo también se pretende adquirir conocimientos complementarios a los ya obtenidos a lo largo de los estudios de Ingeniería en Informática relativos a las tecnologías de procesamiento XML, así como a las herramientas de construcción de procesadores de lenguaje.

1.3 Estructura del documento

La estructura de esta memoria es la siguiente:

- En el Capítulo 2 se realiza una introducción a los conceptos y tecnologías más relevantes en el proyecto, así como una breve introducción a las herramientas existentes y las utilizadas en el proyecto.

- En el Capítulo 3 se describe el entorno XLOP a un nivel enfocado al usuario. En este capítulo se describe el lenguaje de especificación de XLOP, así como el uso del entorno y su ejecución, mediante la realización de un ejemplo sencillo.

- En el Capítulo 4 se detalla la arquitectura e implementación del entorno XLOP. En este capítulo se realiza un estudio detallado de los conceptos y los algoritmos que se han implementado en el proyecto.

- En el Capítulo 5 se describe el desarrollo de la aplicación <e-Tutor> . En este capítulo, se detalla la arquitectura lógica de <e-Tutor> y los pasos de construcción de la aplicación mediante el entorno XLOP. Por motivos de simplicidad, la discusión omite algunas características de <e-Tutor> que resultan similares a otras presentadas. La versión completa se incluye en un apéndice.

- Por último, el Capítulo 6 presenta las principales conclusiones obtenidas con el desarrollo de este trabajo, así como propone posibles extensiones y trabajos futuros.

Se incluyen también dos apéndices:

- El Apéndice A incluye los elementos correspondientes a la versión completa del sistema <e-Tutor>.

- El Apéndice B incluye la gramática del lenguaje de especificación de XLOP.

1.4 Contexto del proyecto

Este trabajo ha sido realizado en el contexto del grupo de investigación en Ingeniería del Software y e-Learning <e-UCM> del Departamento de Ingeniería del Software e Inteligencia Artificial de la Universidad Complutense de Madrid. Más concretamente, el trabajo se enmarca en el proyecto de investigación "Tecnologías de Marcado Descriptivo -XML- como base a un Proceso de Desarrollo de Software Guiado por Lenguajes", proyecto Santander/UCM PR34/07-15865 financiado por la Universidad Complutense y el Banco Santader, y dirigido por el Prof. José Luis Sierra Rodríguez. La concepción de XLOP, así como las bases conceptuales y teóricas en las que se basa el sistema, son fruto del trabajo que se está desarrollando en la Tesis Doctoral del Prof. Antonio Sarasa Cabezuelo "Procesamiento de Documentos XML mediante Herramientas de Construcción de Procesadores de Lenguaje".

Capítulo 2
Revisión de Conceptos y Tecnologías

2.1 Introducción

En este capítulo se presenta una introducción a los aspectos conceptuales y tecnológicos más importantes que tienen relación directa con el proyecto, así como una introducción a las herramientas software más importantes que se han utilizado en el mismo. Primero, se presenta una breve introducción a los lenguajes de marcado, destacando XML como el lenguaje de marcado más utilizado en la actualidad, y que será el tratado por nuestro proyecto. A continuación, se diferencian los tipos de marcos genéricos para el procesamiento de los documentos XML. Seguidamente, se examinan las características de las herramientas existentes que permiten construir procesadores de lenguaje. Por último, se dedica una breve introducción a los conceptos más característicos de las gramáticas de atributos y su relación con XLOP.

2.2 Lenguajes de marcado y XML

2.2.1 Lenguajes de marcado

Los lenguajes de marcado permiten codificar un documento de manera que, junto con el texto, se incorporan etiquetas o marcas que aportan información adicional acerca de la estructura del documento o de su presentación [Coombs et al. 1987]. La idea de marcado siempre ha estado muy relacionada con la presentación de la información. A lo largo del tiempo, los procesadores de texto han ido evolucionando para permitir presentaciones más sofisticadas y elegantes. Estos procesadores introducen marcado en sus documentos para indicar qué tipo de acción deben realizar con el texto que sigue (p.ej., un cambio de fuente o un cambio de línea). Sin embargo, estas marcas o etiquetas quedan ocultas al usuario del procesador de texto, que sólo vé el texto formateado. Hoy en día, a los procesadores de texto se les pide mucho más que mostrar textos con distintas fuentes. Estas nuevas necesidades han hecho que los procesadores de texto cambien el uso que dan al marcado, pasando a utilizarlo para realizar cada vez procesos más complejos.

Los lenguajes de marcado son de diferente naturaleza a los lenguajes de programación. Mientras que los lenguajes de programación están orientados a describir programas informáticos, y son utilizados por programadores, los lenguajes de marcado se han utilizado tradicionalmente, y se siguen utilizando, en la industria editorial y de la comunicación. Sus usuarios no son expertos en informática, sino que son autores, editores e impresores.

Dependiendo de la funcionalidad dentro de un documento, el marcado se puede dividir en varios tipos [Coombs et al. 1987]. Cada uno de estos tipos de marcado presenta ventajas y desventajas desde el punto de vista del intercambio de información.

2.2.1.1 Marcado presentacional y puntuacional

El marcado presentacional es útil para aumentar la legibilidad de un documento. Ejemplos típicos de marcado presentacional son los espacios en blanco y el indentado. Este tipo de marcado aumenta la legibilidad de los datos, pero resulta insuficiente para soportar el procesamiento automático de la información. Sirva como ejemplo, “Estoesuntexto” y “Esto es un texto”: aunque ambos textos contienen la misma información, el segundo es más legible. Los espacios en blanco son marcas presentacionales que aumentan la legibilidad sin cambiar el significado de los contenidos.

El marcado de puntuación ayuda a la correcta interpretación de los contenidos. Por ejemplo, la interpretación de “Esto es un texto” cambia si se introducen los símbolos de interrogación: “¿Esto es un texto?”. Los marcados de presentación y de puntuación están totalmente extendidos en la creación de documentos, hasta el punto de que se pueden considerar intrínsecamente unidos a los contenidos. Esto hace que también se conozca a estos tipos de marcado como marcados implícitos.

2.2.1.2 Marcado procedimental

El marcado procedimental consiste en marcas que establecen la realización de unas determinadas acciones sobre los contenidos. Está orientado a realizar presentaciones del texto sobre diferentes formatos (por ejemplo, impresión en papel), está formalizado y perfectamente distinguido de los contenidos. Debido a que el programa que procesa un documento marcado procedimentalmente debe interpretar el código en el mismo orden en el que aparece, para formatear un texto, debe existir una serie de órdenes inmediatamente antes del texto en cuestión. Esto indicará los efectos a aplicar sobre el texto. Inmediatamente, después del texto deberán existir órdenes inversas que reviertan dichos efectos. Por ejemplo, en los procesadores de texto se puede indicar que un texto debe mostrarse en **negrita**, en *cursiva*, o con un tamaño de fuente más grande que el habitual. En sistemas más avanzados se utilizan macros o pilas que facilitan el trabajo. Algunos ejemplos de sistemas de marcado procedimiental son troff [Emerson 1986], TeX [Seroul et al. 2008], y PostScript [Adobe 2007] .

2.2.1.3 Marcado descriptivo

El marcado descriptivo permite realizar una separación entre la estructura lógica de los documentos y su procesamiento, a fin de resolver las desventajas que presenta el marcado procedimental. Las etiquetas se utilizan para describir los fragmentos de texto sin especificar la

manera en que estos han de ser finalmente presentados. Este tipo de marcado aparece, por ejemplo, al indicar que un determinado fragmento de texto en un libro es un título, un párrafo, o el nombre de un autor. El marcado descriptivo es flexible debido a que permite realizar cambios sin alterar el esquema de marcado. Es sencillo, debido a que simplifica la tarea de reformatear un texto, puesto que la información del formato está separada del propio contenido. Así mismo, presenta un carácter atributivo, al reflejar la estructura lógica de los contenidos en lugar de especificar la forma de procesarlos. Esto permite tener claramente identificados a los integrantes de las distintas categorías lógicas del documento [Goldfarb 1991]. Los lenguajes expresamente diseñados para generar marcado descriptivo son SGML (S*tandard* G*eneralized* M*arkup* L*anguage*) [Goldfarb 1991] y XML (*eXtensible* M*arkup* L*anguage*) [Bray et al. 2008].

2.2.1.4 Marcado referencial y metamarcado

El marcado referencial permite referir entidades que almacenan contenidos, a su vez, posiblemente marcados. Este tipo de marcado surge para facilitar el mantenimiento de documentos complejos, permitiendo por ejemplo, referir fragmentos de un documento en un documento maestro.

El metamarcado permite definir el uso y aplicación de otro marcado. Este tipo de marcado permite definir las reglas de combinación que debe seguir cierto sistema de marcado descriptivo (por ejemplo, dentro de un título no se admite un párrafo), definir los referentes válidos utilizados posteriormente en el marcado referencial, o abstraer procedimientos que, posteriormente, se utilizan para procesar contenidos en un sistema procedimiental.

2.2.2 XML

Como se ha indicado anteriormente, XML son las siglas de Extensible Markup Language, un lenguaje de marcado desarrollado por el World Wide Web Consortium (W3C) para el marcado de contenidos en Internet. Es una simplificación y adaptación de SGML orientada a facilitar el uso mediante la eliminación de las características más específicas y problemáticas que éste último presenta. XML se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas. No es realmente un lenguaje en particular, sino que permite definir la gramática de lenguajes de marcado específicos, así como establece las reglas genéricas de aplicación del marcado a los documentos. XML es una tecnología sencilla que permite compartir la información entre sistemas de una manera fácil, segura y fiable. Aparte de la propia especificación [Bray et al. 2008], algunas referencias útiles sobre XML son [Bradley, 2001; Birbeck et al, 2001; Maruyama et al. 2002].

2.2.2.1 Estructura de un documento XML

La tecnología XML pretende establecer una manera estructurada, más abstracta y lo más reutilizable posible para expresar la información. La información se mantiene estructurada al componerse de conjuntos bien definidos de información, que a su vez, pueden componerse de subconjuntos de información. De esta manera se obtiene la información estructurada en forma de árbol a través de las marcas de etiquetas que definen fragmentos de información con un sentido claro y definido. XML surge en 1998 como fruto del consorcio W3C como metalenguaje para el intercambio de información en internet. Su desarrollo se basa en la simplificación del SGML extrayendo un subconjunto de especificaciones que aseguran la mayoría de las cualidades del metalenguaje con la menor complejidad posible. En la actualidad, XML se ha convertido en un estándar para el intercambio de información entre aplicaciones que permite asegurar la validez de los documentos mediante la incorporación explícita de una DTD (Document Type Definition).

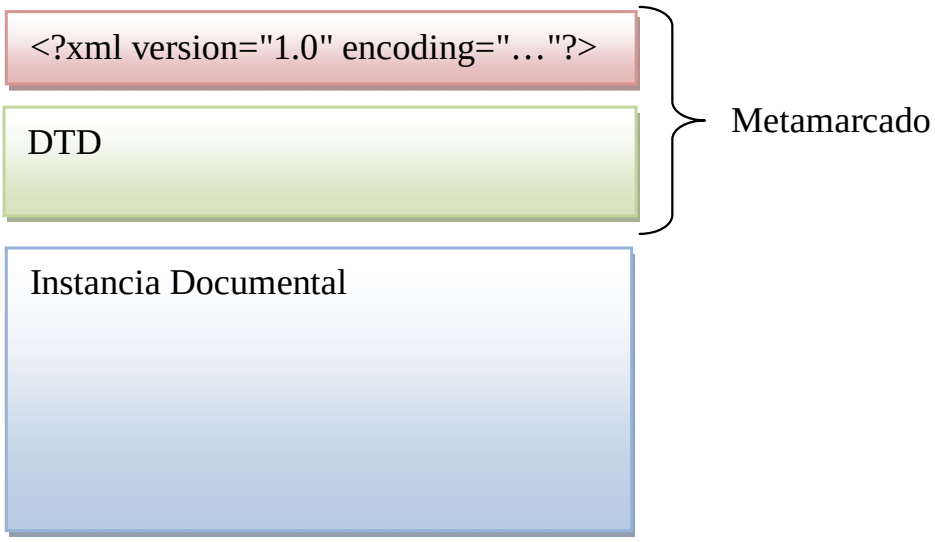

Figura 2.2.1. Estructura de un documento XML.

En la Figura 2.2.1 se presenta la estructura de un documento XML. En ella se pueden diferenciar claramente tres partes:

- El encabezado del documento se compone de metainformación que va dirigida a los procesadores XML y no se considera parte de la información del documento. Al comienzo del documento se declara la versión XML y el conjunto de caracteres utilizado. Más concretamente, XML obliga a los sistemas de procesamiento a soportar como mínimo, Unicode en sus codificaciones UTF-8 y UTF-16 [Bradley, 2001].

- La estructura del documento se especifica a través de una DTD. Las DTDs en XML son una simplificación de las DTDs SGML y permiten comprobar y validar la estructura de los documentos XML.

- La instancia documental contiene la información del documento, organizada en una estructura jerárquica mediante el uso de elementos o partes de información contenidas entre etiquetas de apertura y cierre.

La Figura 2.2.2 muestra un ejemplo de documento XML con cada una de las tres partes distinguidas en la Figura 2.2.1.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE Recetas SYSTEM "Recetas.dtd" [<!ELEMENT Recetas (Receta)*>]>

<Recetas>
     <Receta tipo=”carnes”>
          <Plato>Nombre del plato</Plato>

          <Ingredientes>
               <Ingrediente>Ingrediente 1</Ingrediente>
               <Ingrediente>Ingrediente 2</Ingrediente>
          </Ingredientes>

          <Pasos>
               <Paso>Primer paso a realizar</Paso>
               <Paso>Segundo paso a realizar</Paso>
               <Paso>Tercer paso a realizar</Paso>
          </Pasos>
     </Receta>
</Recetas>
```

Figura 2.2.2. Ejemplo de documento XML

Los documentos XML organizan la información en unidades o *elemento*s que dividen la información en categorías lógicas. Los elementos son información contenida entre marcas establecidas por una etiqueta de apertura y su correspondiente etiqueta de cierre (por ejemplo, <Recetas> y </Recetas> de la Figura 2.2.2). En XML, al igual que en SGML, es posible asociar con los elementos pares *atributo-valor* para indicar metainformación adicional asociada con los mismos, o para expresar relaciones adicionales entre elementos. La etiqueta de apertura <Receta> de la Figura 2.2.2 muestra un ejemplo de *atributo-valor* para indicar información adicional sobre el tipo de una receta.

Mediante una DTD se establecen los elementos o las reglas que deben cumplir los documentos XML para garantizar que presentan la misma estructura lógica.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!ELEMENT Receta (Plato, Ingredientes, Pasos)>
        <!ELEMENT Plato (#PCDATA)>
        <!ATTLIST Plato tipo CDATA #IMPLIED>

         <!ELEMENT Ingredientes (Ingrediente)>
                 <!ELEMENT Ingrediente (#PCDATA)>

         <!ELEMENT Pasos (Paso)>
                 <!ELEMENT Paso (#PCDATA)>
```

Figura 2.2.3. Ejemplo de DTD de la Figura 2.2.2.

Estos archivos DTD permiten garantizar que los documentos XML siguen una estructura y formato especifico y válido para asegurar su correcto procesamiento. La Figura 2.2.3 muestra la DTD que presenta la estructura lógica de las instancias XML para cada receta.

Es interesante conocer que XML permite disponer de documentos que no incluyen una DTD. Sin embargo, dichos documentos sin DTD deberán seguir la estructura que el procesador de documentos XML requiere, es decir, deberán estar “bien formados”. Los documentos denominados como "bien formados" son aquellos que cumplen con todas las definiciones básicas de formato (siendo la más importante el correcto anidamiento de las etiquetas) y pueden, por lo tanto, analizarse correctamente con cualquier analizador sintáctico que cumpla con la norma.

En el Capítulo 3 y el Capítulo 5 se muestra la construcción de ejemplos de aplicaciones en XLOP junto a la especificación de los documentos XML y la importancia que implica el disponer de DTDs para diseñar y llevar a cabo el correcto procesamiento de dichos documentos.

2.2.3 Procesamiento de Documentos XML

Siguiendo las directrices del marcado descriptivo [Coombs et al. 1987], XML no norma la forma de procesar los documentos marcados. Para llevar a cabo el procesamiento es necesario, por tanto, otros medios. Para ello pueden emplearse lenguajes específicos, que abordan algunos tipos de procesamiento. Por ejemplo, XSLT (*eXtensible Stylesheet Language Transformations*) [Kay 2007] es un lenguaje orientado a la transformación de documentos XML en documentos XML, que constituye un buen ejemplo de esta tendencia.

No obstante, para la realización de tareas más complejas y abiertas, es necesario utilizar marcos de procesamiento XML más genéricos [Lam et al. 2008]. Estos marcos ofrecen típicamente funcionalidades básicas para realizar el procesamiento. Entre estas funcionalidades destacan funcionalidades de lectura (a través de componentes denominados *parsers* XML), comprobación de buena formación, y validación de documentos XML, así como exposición de los documentos leidos en un formato apropiado para su procesamiento. Los tipos más usuales de marcos de procesamiento genéricos para XML son los marcos *orientados a árboles*, y los *orientados a eventos*.

Los marcos orientados a árboles promueven la creación de estructuras arbolescentes en base al contenido del documento. Estos marcos incluyen *parsers* capaces de transformar los documentos en árboles. Los árboles permiten el acceso inmediato a cualquier parte de la información en cualquier momento, al contrario de lo que sucede en los marcos orientados a eventos, donde el acceso a la información es secuencial. El alojamiento de toda la información en memoria permite realizar cálculos complejos, así como usar programación recursiva, a diferencia de los marcos orientados a eventos, donde suele ser preciso guardar la información en variables o incluso realizar dos pasadas sobre el documento. Sin embargo, el alojamiento de toda la información para documentos muy grandes implica un elevado consumo de recursos y mayor tiempo de proceso en la construcción del árbol, por lo que son menos eficientes en tiempo y espacio que los marcos orientados a eventos.

Los marcos orientados a eventos recorren los documentos de manera secuencial disparando un evento cada vez que se reconoce un componente. Para el uso de este tipo de marcos sólo es necesario codificar los tratamientos de eventos que se disparan al reconocer un tipo determinado de componente y realizar la gestión de la información de dicho componente, información que se obtiene como parámetro del evento correspondiente. La principal ventaja de los marcos orientados a eventos es que son sencillos de programar y rápidos en procesamiento, debido a que sólo disparan eventos al ir recorriendo el documento. Al no almacenar la información en estructuras internas, requieren pocos recursos durante el procesamiento. Esto hace interesante su utilización para procesar grandes documentos. Sin embargo, para procesamientos complejos, puede ser necesario mantener la información en una estructura de rápido acceso, objetivo que no cumplen estos marcos.

De esta forma, la elección de un tipo u otro de marco radica en el proceso a realizar y el tamaño de los documentos a tratar. Si el proceso es complejo y el tamaño de los documentos no es elevado, el uso de un marco orientado a árboles puede ser mucho más interesante que el uso de un marco orientado a eventos. Un compromiso intermedio es el proporcionado por los denominados *marcos* pull, de los cuáles StAX es la especificación más significativa [Lam et al. 2008].

A continuación se examina brevemente las características de los marcos de procesamiento XML más ampliamente utilizados: DOM y SAX.

2.2.3.1 Un marco orientados a árboles: DOM

Es importante disponer de algún mecanismo estándar para acceder a los árboles construidos por los marcos orientados a árboles a fin de poder independizar la lógica de la aplicación del tipo de marco utilizado. DOM es el estándar más comúnmente utilizado para la creación y modificación de estos árboles documentales [DOM 2009].

DOM son las siglas de *Document Object Model*. Ha surgido como fruto de los esfuerzos dirigidos por el consorcio W3C. Su especificación aporta un conjunto de interfaces estándar para la construcción, acceso y modificación dinámica de contenido estructurado en árboles. Se

caracteriza por ser esencialmente una interfaz de programación de aplicaciones que proporciona un conjunto estándar de objetos para representar documentos HTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos. DOM está pensado para su uso en lenguajes orientados a objetos como C++ o Java. El modelo estructural que establece para árboles de documentos XML no indica cómo dichos árboles deben ser físicamente representados.

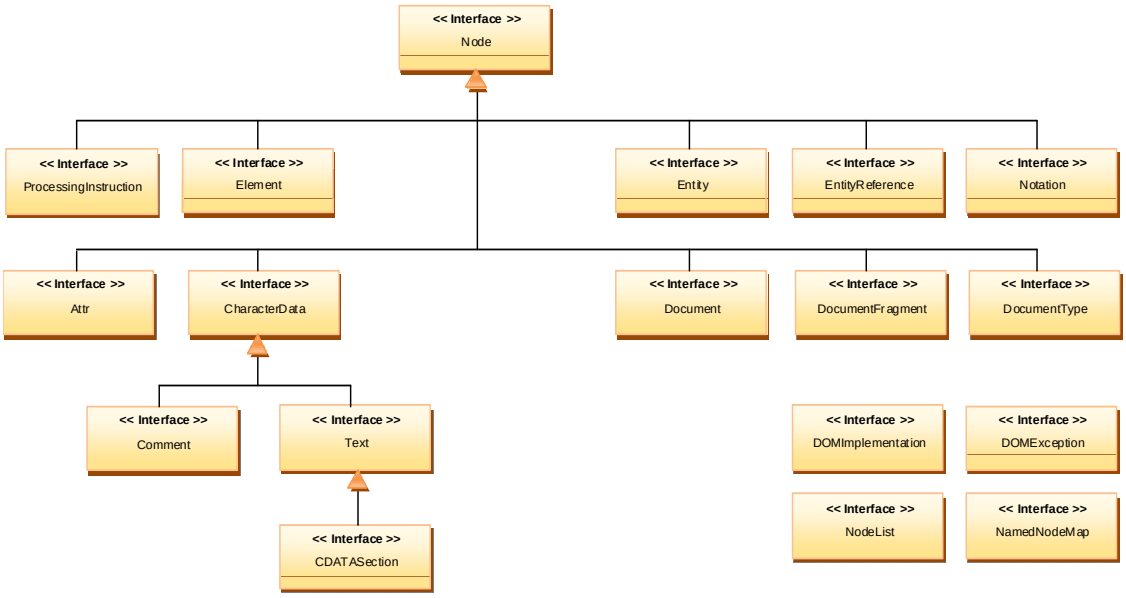

Figura 2.2.4. Diagrama de clases de la estructura DOM de nivel 1.

Existen varias especificaciones de la arquitectura clasificada en niveles con el fin de integrar facilidades adicionales. Mientras que el nivel 1 se limita a capturar el modelo estructural básico introducido por la especificación, el nivel 2 en DOM introduce características tales como espacios de nombre o propagación de eventos. También se ha incorporado un nivel 3 en DOM que hace uso de la DTD y la validación de documentos.

La Figura 2.2.4 muestra la arquitectura DOM de nivel 1, que caracteriza la estructura básica de los árboles documentales. La estructura de DOM consta de árboles de nodos cuyo tipo depende del contenido. Los nodos pueden ser Document, Element, Attr, Text, Comment, etc., tal y como muestra la Figura 2.2.4. A modo de ejemplo, los nodos de tipo Text llevan asociados como valor el contenido del texto, mientras que los nodos de tipo Element llevan asociados una secuencia de nodos hijos y una tabla de atributos. Esta interfaz proporciona un conjunto de métodos específicos que permiten acceder y modificar los valores de los atributos del elemento. Cabe destacar la interfaz Document, que centraliza la creación de los diferentes nodos mediante métodos factoría que permiten realizar implementaciones concretas sobre el resto de los elementos, así como especializar adecuadamente una implementación de DOM. Mediante estas interfaces, se establece una relación entre los nodos que permite recorrer la

estructura iterando sobre la secuencia de los nodos hijos de los elementos, o bien aprovechando las relaciones especiales entre nodos.

En resumen, DOM proporciona una interfaz estándar que permite independizar las aplicaciones de las representaciones concretas de los documentos, así como conectar fácilmente distintas aplicaciones entre sí.

2.2.3.2 Un marco orientado a eventos: SAX

En el proyecto se utiliza un parser orientados a eventos, en especial se ha realizado la implementado de eventos que proporcionan las interfaces del parser SAX, parser que ofrece un rápido procesamiento secuencial de documentos y un bajo consumo de recursos, objetivo que se adapta a la perfección al de XLOP.

SAX son las siglas de S*imple* API *for* XML. SAX es la arquitectura estándar para la lectura y procesamiento de documentos XML siguiendo el modelo orientado a eventos [Brownell 2002]. Aunque su desarrollo no ha sido controlado por el W3C, ha contado con su respaldo. SAX existe en diferentes versiones. La versión 1.0 se propuso en 1998 y se mejoró con inclusión de diversas extensiones, como el soporte para espacios de nombres, en la versión 2.0. en el 2002. SAX define un API orientado a objetos que requiere la configuración específica del tratamiento de eventos asociados al procesamiento XML.

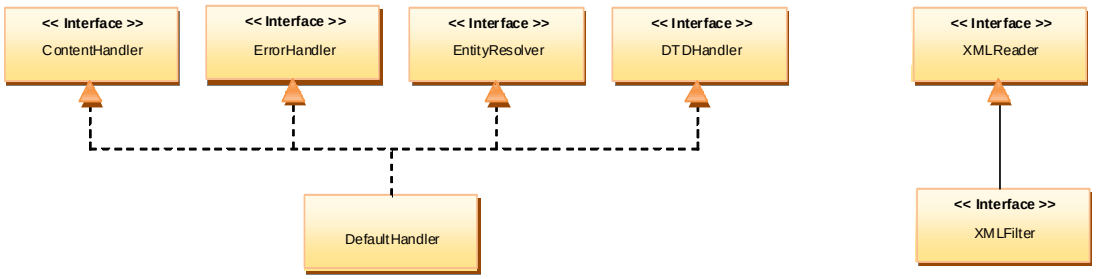

Figura 2.2.5. Diagrama de clases de la estructura SAX.

La Figura 2.2.5 muestra las diferentes interfaces que permiten gestionar los eventos clasificados según el tipo de evento que disparan. Más concretamente:

- DocumentHandler: Se caracteriza por ser la principal interfaz, cuyos métodos se invocan a medida que se reconocen los distintos tipos de nodos en el procesamiento del documento.
- DTDHandler: Interfaz que dispone de métodos que se invocan para gestionar los eventos relacionados con la DTD.
- ErrorHandler: Interfaz cuyos métodos se invocan para notificar diferentes tipos de errores durante el procesamiento del documento.

- EntityResolver: Interfaz que permite resolver los identificadores del sistema e identificadores públicos asociados con entidades a los sitios reales donde dichas entidades están almacenadas.

- XML Reader: Representa la funcionalidad básica que deben seguir los parsers que leen documentos y generan eventos SAX.

- XMLFilter: Interfaz que extiende a XMLReader para facilitar la composición de procesadores SAX. Las implementaciones de XMLFilter permiten construir un nuevo procesador SAX a partir de otro procesador SAX o procesador padre.

La clase DefaultHandler proporciona una implementación estándar para las cuatro primeras interfaces. En el proyecto, la implementación de dichas interfaces se realiza partiendo de la implementación estándar que proporciona la clase DefaultHandler. En el Capítulo 4 sección 4.9 se detalla la implementación SAX realizada para la gestión de los diferentes nodos de los documentos XML.

2.3 Herramientas de construcción de procesadores de lenguaje

Las herramientas que permiten generar procesadores de lenguaje (o componentes de procesadores de lenguaje, como los analizadores sintácticos) permiten al programador trabajar a un nivel de abstracción más elevado que el que deberían usar si se partiese desde cero.

Los generadores de procesadores de lenguaje más populares son los generadores de analizadores sintácticos [Bennet 1990]. El análisis sintáctico, o *parsing*, consiste en el análisis de una secuencia de símbolos a fin de determinar su estructura gramatical con respecto a una gramática formal dada [Grune&Jacobs 2008]. Como resultado de este proceso se genera un árbol de análisis sintáctico. Existen dos tipos principales de analizadores sintácticos en base al conjunto de gramáticas que soportan. Estos son los *analizadores descendentes* y los *analizadores ascendentes* [Aho et al. 2007]. Los analizadores descendentes construyen el árbol de análisis sintáctico de arriba a abajo, desde la raíz hasta las hojas, mientras que los analizadores ascendentes lo construyen de abajo a arriba, desde las hojas hasta concluir en la raíz. Si bien los métodos ascendentes permiten manejar una gama más amplia de gramáticas, los métodos descendentes son más fáciles de implementar.

2.3.1 Generadores de traductores

Generalmente las etapas de análisis léxico y sintáctico son soportadas por distintas herramientas de generación automática, lo que permite independizar las funciones y obtener mayor flexibilidad. Como herramientas de generadores de analizadores sintácticos en Java que utilizan métodos *descendentes* podemos encontrar JavaCC, J*ava* C*ompiler* C*ompiler*, desarrollada por Sun Microsystems en 2000 [Kodaganallur 2004], y ANTLR, A*nother* T*ool* for

L*anguage* R*ecognition*, desarrollada por Parr en 1995 [Parr 2007]. Como herramentas Java de generación de analizadores que utilizan métodos *ascendentes* podemos encontrar SableCC, desarrollado por Gagnon y Hendren en 1998 [Gagnon 1998], y CUP, C*onstructor* of U*seful* P*arsers*, desarrollado por Hudson en 1997 [Appel 1997; Hudson 1999]. Véase también [Gajardo&Mateu 2004] para más detalles.

Los métodos de análisis ascendente son más potentes porque sus gramáticas no requieren demasiadas modificaciones para que puedan ser utilizadas como base para la construcción del analizador, como en el caso de los métodos descendentes. Sin embargo, la incorporación de *acciones semánticas* es substancialmente más complicada en los analizadores ascendentes que en los descendentes [Purdom&Brown 1980; Aho et al. 2007]. En el proyecto se usa la tecnología JavaCC para la creación del analizador sintáctico / traductor del lenguaje de especificación XLOP, y la herramienta CUP como notación en la que se expresa los *parsers* de las aplicaciones que se construyen, y que se generan automáticamente a partir de las especificaciones XLOP.

Herramientas vs características	CUP	SableCC	JavaCC	ANTLR
Método de análisis ascendente	√	√	X	X
Integración	X	√	√	√
Reglas de desambiguación	√	X	√	√
Asociatividad de acciones a producciones	√	X	√	√
Soporta código en lenguaje Java	√	√	√	√
Disponibilidad del código fuente	√	√	X	√
Su código puede ser modificado por el programador	√	√	X	√
Documentación y soporte	√	√	√	√

Figura 2.3.1. Tabla comparativa de características de los analizadores sintácticos mencionados (fuente [Gajardo&Mateu 2004])

La Figura 2.3.1 muestra una tabla comparativa que refleja las diferencias y características de las cuatro herramientas mencionadas como guía para la elección de una herramienta u otra. Es importante destacar que las cuatro herramientas analizadas son de código abierto con posibilidad de modificación de código por parte del programador, excepto JavaCC que no provee su código fuente.

A continuación se analizan con más detalle JavaCC y CUP, al ser éstas las herramientas que intervienen en el proyecto.

2.3.2 JavaCC

JavaCC son las siglas de J*ava* C*ompiler* C*ompiler*. Es un generador de analizadores sintácticos de código abierto para el lenguaje de programación Java. JavaCC fue desarrollado por Sun Microsystems aunque en la actualidad es mantenido por MetaMata. JavaCC permite generar un parser, en código Java, para una gramática presentada en notación EBNF. El parser que genera se caracteriza por ser un analizador de tipo descendente para la clase de gramáticas LL(K) [Aho et al. 2007]. La Figura 2.3.2 muestra el código de un *parser* implementado en JavaCC.

```
options  {
  STATIC=false;
}
PARSER_BEGIN(Sumador)
   class Sumador {
      public static void main(String[] args) throws ParseException {
         Sumador sumador = new Sumador(System.in);
         sumador.listaExp();
      }
   }
PARSER_END(Sumador)

TOKEN: {
   <#dpos:["1"-"9"]> |
   <#dig:<dpos>|"0"> |
   <num:<dpos>(<dig>)*|"0">
}
SKIP: {<["\t"," ","\r","\b","\n"]>}

void listaExp() : {int v;} {
   v = exp() {System.out.println("Suma = "+v);} rlistaExp()
}
void rlistaExp() : {} {
   "," listaExp() | "." <EOF>
}
int exp() : {int vh; int v;} {
   vh = fact() v = rexp(vh) {return v;}
}
int rexp(int vh) : {int vt,v;} {
   "+" vt = fact() v = rexp(vh + vt) {return v;}
   | "-" vt = fact() v = rexp(vh - vt) {return v;}
   | {return vh;}
}
int fact() : {Token t;int v;} {
    t = <num> {return Integer.valueOf(t.image).intValue();}
}

/***** Ejemplo de ejecución *****
5 + 6 - 1,
Suma = 10
3 + 4 - 7.
Suma = 0
*******************************/
```

Figura 2.3.2. Ejemplo de sumador en JavaCC.

Como sugiere la Figura 2.3.2, la sintaxis utilizada en el lenguaje de especificación de JavaCC es muy similar a Java. De hecho, el código asociado a las reglas JavaCC es código Java. Como sugiere la Figura 2.3.2, los ficheros JavaCC están estructurados en varias secciones: una sección donde se declara la clase y los métodos auxiliares que se utilizan en las reglas, junto a la manera de inicializar el proceso de análisis / traducción, otra sección donde se determina la manera de reconocer los símbolos del archivo, y la sección principal donde se declaran las reglas que se aplicarán durante el proceso de análisis / traduccion. En el Capítulo 4 sección 4.5.1 se analizará el uso de la herramienta JavaCC en el desarrollo del entorno XLOP.

2.3.3 CUP

Como se ha indicado anteriormente, CUP es un generador de traductores ascendentes de código abierto para el lenguaje de programación Java. CUP soporta la clase de gramáticas LALR(1) [Aho et al. 2007]. La Figura 2.3.3 muestra el contenido de un parser implementado en CUP.

```
// CUP specification for a simple expression evaluator (w/ actions)

import java_cup.runtime.*;

/* Preliminaries to set up and use the scanner.  */
init with {: scanner.init();              :};
scan with {: return scanner.next_token(); :};

/* Terminals (tokens returned by the scanner). */
terminal           SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal           UMINUS, LPAREN, RPAREN;
terminal Integer   NUMBER;

/* Non-terminals */
non terminal            expr_list, expr_part;
non terminal Integer    expr;

/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

/* The grammar */
expr_list ::= expr_list expr_part | expr_part;

expr_part ::= expr:e {: System.out.println("= " + e); :} SEMI;

expr      ::= expr:e1 PLUS expr:e2 {: RESULT = new Integer(e1.intValue() +
                                        e2.intValue()); :}
          |   expr:e1 MINUS expr:e2 {: RESULT = new Integer(e1.intValue() -
                                         e2.intValue()); :}
          |   expr:e1 TIMES expr:e2 {: RESULT = new Integer(e1.intValue() *
                                        e2.intValue()); :}
          |   expr:e1 DIVIDE expr:e2 {: RESULT = new Integer(e1.intValue() /
                                        e2.intValue()); :}
          |   expr:e1 MOD expr:e2   {: RESULT = new Integer(e1.intValue() %
                                        e2.intValue()); :}
          |   NUMBER:n              {: RESULT = n; :}
          |   MINUS expr:e          {: RESULT = new Integer(0 - e.intValue());
                                       :} %prec UMINUS
          |   LPAREN expr:e RPAREN  {: RESULT = e; :} ;
```

Figura 2.3.3. Ejemplo de sumador/multiplicador en CUP (ejemplo tomado de [Hudson 1999]).

La Figura 2.3.3 presenta la construcción de un sumador/multiplicador mediante un archivo de especificación en CUP. Se pueden contemplar el parecido similar que presentan sus archivos de especificación con respecto a los de JavaCC (Figura 2.3.2). No obstante, no debe olvidarse que la arquitectura de *parser* subyacente es totalmente diferente: análisis ascendente vs. análisis descendente. En el Capítulo 4 sección 4.8.1 se analizará con más detalle el papel jugado por la herramienta CUP en el contexto del entorno XLOP.

2.4 Gramáticas de atributos

El lenguaje de especificación de XLOP se basa en el formalismo de las gramáticas de atributos [Knuth 1968; Paaki 1995]. Este formalismo fue propuesto por Donald E. Knuth a finales de los sesenta como un mecanismo para añadir *semántica* a los lenguajes incontextuales.

```
Exp ::= Exp + Term
   Exp0.val = Exp1.val + Term.val
   Exp1.varsh = Exp0.varsh
   Term.varsh = Exp0.varsh

Exp ::= Term
   Exp.val = Term.val
   Term.varsh = Exp.varsh

Term ::= var
   Term.val = valorDe(Term.varsh,var.nombre)

Term ::= num
   Term.val = num.valor
```

Figura 2.4.1. Ejemplo de gramática de atributos.

La Figura 2.4.1 muestra un ejemplo de gramática de atributos. De esta forma, una gramática de atributos consta de:

- Una *gramática incontextual* que caracteriza la sintaxis estructural del lenguaje mediante un conjunto de *reglas sintácticas* o *producciones* (por ejemplo, Exp ::= Exp + Term)

- Un conjunto de *atributos semánticos* añadidos a los símbolos de la citada gramática. Estos atributos pueden ser de dos tipos: atributos *sintetizados* (en el ejemplo, val es un atributo sintetizado) y atributos *heredados* (en el ejemplo, varsh). Los atributos toman valores en los nodos de los *árboles sintácticos* impuestos por la gramática incontextual sobre las sentencias. Los valores de los atributos sintetizados representan la *semántica* de los fragmentos de sentencia que penden de los nodos (en el ejemplo, val representa el valor de las expresiones), mientras que los de los atributos heredados representan información de contexto (en el ejemplo, varsh representa una tabla con el valor

de las variables que aparecen en las expresiones). Los atributos sintetizados en un nodo se computan a partir de los sintetizados de sus hijos y de los propios atributos heredados del nodo. Por su parte, los atributos heredados se computan a partir de los sintetizados de los hermanos y de los heredados del padre.

- Un conjunto de *ecuaciones semánticas* para cada producción. Estas ecuaciones indican cómo computar los valores de los atributos sintetizados de la cabeza y de los atributos heredados de los símbolos del cuerpo. Para ello aplican *funciones semánticas* sobre los atributos utilizados en dicho cómputo. Por ejemplo, Term.val = valorDe(Term.varsh,var.nombre) indica que para encontrar el valor del término cuando éste es una variable, es necesario buscar el valor de la variable en la tabla.

El *axioma* de la gramática también puede tener atributos heredados (en el ejemplo, varsh es un atributo heredado del axioma Exp). Así mismo, los *terminales* también pueden tener atributos sintetizados, que se denominan *atributos léxicos* (p.ej. nombre es un atributo léxico del terminal var). Los valores de estos atributos se fijarán externamente (p.ej. los atributos léxicos se fijarán durante el análisis léxico).

Durante la preparación de una gramática de atributos *no* es necesario especificar explícitamente en qué orden tienen que aplicarse las ecuaciones semánticas para encontrar los valores de los atributos en los árboles sintácticos (es decir, para *evaluar* dichos atributos). De esta forma, las gramáticas de atributos son mecanismos descriptivos de más alto nivel que los *esquemas de traducción* soportados por las herramientas típicas de construcción de procesadores de lenguaje (p.ej. JavaCC, ANTLR, YACC, CUP, etc.), en los que sí es necesario explicitar el orden de ejecución de las *acciones semánticas*. Por el contrario, en una gramática de atributos el orden de evaluación se deriva de las *dependencias* entre los atributos introducidas por las ecuaciones semánticas. El método de evaluación en sí puede ser *estático* o *dinámico* [Ablas 1991]. Los métodos estáticos analizan la gramática durante la generación del evaluador para encontrar un orden de evaluación que funciona para cualquier sentencia. Por su parte, los métodos dinámicos deciden el orden de evaluación para cada sentencia particular. En XLOP se adoptará un método de evaluación dinámica, ya que los métodos dinámicos aceptan una clase más amplia de gramáticas de atributos que los estáticos, aún a consta de una ligera pérdida de eficiencia.

Capítulo 3
El Entorno XLOP

3.1 Introducción

XLOP (*XML Language-Oriented Processing*) es un entorno que utiliza *gramáticas de atributos* para describir cómo procesar documentos XML marcados con un determinado vocabulario.

Las *gramáticas de atributos* son un formalismo declarativo ampliamente utilizado en la descripción de la sintaxis, las restricciones contextuales, y la traducción de los lenguajes informáticos.

XLOP sigue un paradigma *dirigido por lenguajes* en el desarrollo de los programas de procesamiento de documentos XML. De acuerdo con este paradigma, dichos programas se entienden como *procesadores de lenguaje*, y el proceso de desarrollo en sí se entiende como el proceso de construcción y mantenimiento de dichos procesadores. De hecho, utilizando XLOP es posible generar automáticamente tales procesadores a partir de especificaciones de alto nivel expresadas como gramáticas de atributos.

XLOP se ha diseñado para ser integrado con Java. Las *funciones semánticas* que se utilizan en las gramáticas de atributos XLOP se implementan como métodos en Java. De esta forma, XLOP ofrece una flexibilidad comparable a la de los marcos de procesamiento genéricos para XML (p.ej. SAX, DOM, STaX, etc.) y un nivel de usabilidad comparable al de enfoques específicos (p.ej. los ofrecidos por lenguajes de transformación como XSLT).

Es más, XLOP permite estructurar las aplicaciones de proceso de XML en dos capas perfectamente diferenciadas:

- Una capa de lógica específica de la aplicación, que incluye la maquinaria necesaria para soportar la funcionalidad de dicha aplicación (p.ej. un marco de aplicación para la representación interna de un tutor inteligente, un conjunto de clases para el procesamiento de metadatos, etc.).

- Una capa *lingüística* de procesamiento XML *dirigido por la sintaxis*. Esta capa se especifica como una gramática de atributos, especificación que se traduce automáticamente a una implementación ejecutable mediante un *generador* XLOP.

La conexión entre ambas capas se realiza mediante una *clase semántica*, que implementa en Java las funciones semánticas utilizadas en la gramática de atributos XLOP, y que media entre las dos capas anteriores. De esta forma, el modelo de desarrollo dirigido por lenguajes de XLOP propugna la separación explícita de estas dos capas, así como facilita el desarrollo y el

mantenimiento de la capa lingüística, ya que ésta se especifica a un nivel mucho más alto que el conseguido con una implementación directa en Java o en cualquier otro lenguaje de programación. De hecho, el formalismo de las gramáticas de atributos es también de más alto nivel que una descripción basada en *esquemas de traducción*, del tipo de los soportados por otros enfoques al procesamiento de XML dirigido por lenguajes (ej., ANTXR [Stanchfield 2009], un entorno construido sobre la herramienta ANTLR, y RelaxNGC [Kawaguchi 2002], una extensión del *lenguaje de esquema documental* RelaxNG [Vlist 2003] utilizada para especificar esquemas de traducción y que permite la generación automática de traductores recursivos descendentes).

XLOP es fruto de las investigaciones llevadas a cabo en el proyecto de investigación Santander/UCM PR34/07-15865 "Tecnologías de Marcado Descriptivo -XML- como base a un Proceso de Desarrollo de Software Guiado por Lenguajes". XLOP, así como alguna de sus aplicaciones y otros esfuerzos realizados con el enfoque de procesamiento XML orientado a lenguajes, se describen en distintas publicaciones [Sarasa et al. 2008, 2009a-e]. Este proyecto de Sistemas Informáticos se ha encuadrado en dicho proyecto de investigación, dando soporte a las tareas de desarrollo del entorno XLOP. No obtante, es inevitable que los desarrolladores de un producto en un proyecto de I+D contribuyan activamente a las actividades de investigación del proyecto. Este ha sido el caso también en este trabajo de Sistemas Informáticos, tal y como queda reflejado en la aparición como co-autores de los integrantes del grupo en algunas de las publicaciones anteriormente citadas (más concretamente, [Sarasa et al. 2009a, 2009d-e]).

3.2 Un ejemplo de aplicación

La creación de una aplicación XLOP comienza con la especificación de una gramática de atributos. Mediante la gramática se define la manera en la que se procesan los documentos XML. A partir de ahora nos referiremos a este tipo de gramáticas de atributos como *gramáticas XLOP*.

La especificación de una gramática XLOP debe estar contenida en un formato de archivo de texto común con extensión .xlop. Este archivo puede ser creado con cualquier editor de texto. Para que el archivo sea válido, la manera de especificar la gramática y su estructura debe seguir unas pautas concretas.

La Figura 3.2.1 muestra un ejemplo sencillo de gramática XLOP sobre la que se explicará el *lenguaje de especificación*. La aplicación descrita por esta gramática tiene como objetivo modificar un archivo de entrada mediante una serie de instrucciones definidas en un archivo XML.

```
Desc ::= <Desc> Sents </Desc> {
   fileh of Sents = loadFile(path of <Desc>)
   end of Desc = saveFile(·"Exito"·, newpath of <Desc>, file of Sents)
}

Sents ::= Sents Sent{
   fileh of Sents(1) = fileh of Sents(0)
   fileh of Sent = file of Sents(1)
   file of Sents = replace(file of Sent, table of Sent)
}

Sents ::= {
   file of Sents = fileh of Sents
}

Sent ::= <Dir> Insts </Dir> {
   table of Sent = table of Insts
   file of Sent = fileh of Sent
}

Sent ::= Inst {
   table of Sent = makeTable(old of Inst, neW of Inst)
   file of Sent = fileh of Sent
}

Insts ::= Insts Inst {
   table of Insts(0) = addToTable(table of Insts(1), old of Inst, neW of Inst)
}

Insts ::= Inst {
   table of Insts = makeTable(old of Inst, neW of Inst)
}

Inst ::= <I> #pcdata <is/> #pcdata </I> {
   old of Inst = text of #pcdata(0)
   neW of Inst = text of #pcdata(1)
}
```

Figura 3.2.1. Gramática XLOP de un cifrador/descifrador de archivos.

3.3 El lenguaje de especificación de XLOP

El lenguaje de especificación de XLOP es un metalenguaje para describir gramáticas de atributos para lenguajes de marcado definidos mediante XML. Una gramática de atributos en XLOP se define mediante un conjunto de reglas sintácticas a las que se asocian un conjunto de atributos o ecuaciones semánticas (Figura 3.3.1). En esta sección se realiza una descripción narrativa del lenguaje. La gramática del mismo puede encontrarse en el apéndice B.

```
Desc ::= <Desc> Sents </Desc> {
   fileh of Sents = loadFile(path of <Desc>)
   end of Desc = saveFile(·"Exito"·, newpath of <Desc>, file of Sents)
}
```

Figura 3.3.1. Ejemplo de regla de producción.
Sección azul: Definición sintáctica de la regla.
Sección verde: Definición semántica de la regla.

En una regla sintáctica se distinguen dos partes: una parte izquierda o *cabecera de la regla*, y una parte derecha *o cuerpo de la regla*; separadas por "::=". Los distintos elementos que pueden formar parte de la definición sintáctica de una regla son de los siguientes tipos:

- *No terminal*: Los no terminales se especifican mediante nombres siguiendo la misma sintaxis de un identificador Java (ver [Gosling et al. 2005]).

- *Elemento XML*: Son elementos pertenecientes al lenguaje de marcado XML y tienen un formato análogo al que presentan en las instancias de documentos XML, pero sin atributos (ver [Bray et al. 2008] para más detalles). Pueden ser de tres tipos: *etiqueta de apertura* (ej. <Desc>), *etiqueta de cierre* (ej. </Desc>) o *etiqueta vacía* (ej. <Desc/>).

- *Texto XML*: Denotan fragmentos de texto en el documento XML. Se declaran mediante "#pcdata".

La cabeza de la regla únicamente puede estar formada por un elemento No terminal. El cuerpo de la regla se compone de un conjunto de dichos elementos. Las reglas de producción vacías o reglas-lambda se declaran sin elementos en el cuerpo de la regla.

La definición semántica de una regla se compone de un conjunto de ecuaciones y se especifica entre llaves "{" y "}". En una ecuación se distinguen dos partes: una parte izquierda o *referencia a atributo* y una parte derecha o *expresión semántica*, separadas por "=". Intuitivamente, estas ecuaciones pueden verse como asignaciones: *"el cálculo perteneciente a la parte derecha de la ecuación se asigna al atributo declarado en la parte izquierda de la ecuación"*. Los distintos elementos que pueden formar parte de la definición semántica de una regla son:

- *Referencia a atributo*: Expresa el atributo perteneciente a un no terminal específico de la definición sintáctica de la regla. Los atributos, al igual que los no terminales, se especifican mediante nombres siguiendo la misma sintaxis que un identificador Java. La referencia a atributo se especifica mediante: un atributo, seguido de "of", seguido de un no terminal y, opcionalmente, seguido de un número de ocurrencia entre paréntesis. Es necesario especificar un número de ocurrencia cuando existen varios no terminales con el mismo nombre del de la referencia en la definición sintáctica de la regla. El número de ocurrencia identifica el no terminal mediante un número (comenzando en cero) y contando el número de veces que aparece repetido el elemento a la izquierda de su posición (véase última regla de la Figura 3.2.1).

- *Función semántica*: Las funciones refieren a métodos implementados en una clase Java, que se denomina la *clase semántica* de la aplicación. Se especifican de la siguiente manera: nombre de la función y, entre paréntesis, los distintos argumentos de la función, separados por coma. El tipo y número de argumentos debe coincidir obligatoriamente con el tipo y número de argumentos del método

al que refiere la función en la clase semántica. Los argumentos pueden ser de dos tipos: referencia a atributo o función (véanse las ecuaciones de la Figura 3.3.1).

- *Valor*: Se especifican entre puntos "·" y permiten introducir valores concretos en las ecuaciones. Los valores no están tipados (véase valor "Éxito" de la Figura 3.3.1, valor que se reconoce como tipo String de Java al haberse introducido las comillas).

La parte izquierda de la ecuación únicamente puede estar formada por una referencia a un atributo. La parte derecha de la ecuación se compone únicamente de un elemento semántico.

Recuerde:

- El *axioma* de la gramática o primera regla de producción a evaluar, se declara situándola al comienzo del fichero .xlop.
- Es posible escribir comentarios de dos maneras:
  - Comentario simple "//": Ignora todos los caracteres leídos desde este punto hasta un retorno de carro.
  - Comentario múltiple entre llaves "/*" y "*/": Ignora todos los caracteres leídos delimitados entre dichas llaves.
- La definición semántica de una regla siempre debe estar contenida entre llaves "{" y "}". Es más, estas llaves siempre deben escribirse aunque no se inserte ninguna ecuación.
- Número de ocurrencia: Sólo es necesario especificarlo cuando se definen varios elementos iguales en la definición sintáctica de una regla. Esto permite identificar al elemento deseado. El número se escribe entre paréntesis e inmediatamente después del nombre del elemento. El número se cuenta a partir de cero y de izquierda a derecha: cuenta el número de veces que se repite dicho elemento en la definición sintáctica de la regla. El número cero es la primera aparición del elemento sintáctico. No especificar un número de ocurrencia equivale a referirse a su primera aparición.

3.3.1 Atributos en las ecuaciones semánticas

No es necesario indicar explícitamente qué atributos son heredados y cuáles son sintetizados, debido a que esta información se deduce del uso de los atributos en la gramática.

Los atributos heredados se definen de la siguiente manera:

- Si para un atributo de un no terminal que pertenece al cuerpo de una regla de producción se define una ecuación de asignación a dicho atributo, entonces el atributo se considerará atributo heredado para dicho no terminal.

- Si un atributo de un no terminal que pertenece a la cabeza de una regla de producción se emplea en las partes derechas de las ecuaciones, el atributo se considerará atributo heredado para dicho no terminal.

Los atributos sintetizados se determinan en base a las siguientes reglas:

- Si para un atributo de un no terminal que pertenece a la cabeza de una regla de producción se define una ecuación de asignación a dicho atributo, entonces el atributo se considerará sintetizado para dicho no terminal.

- Si un atributo de no terminal que pertenece al cuerpo de una regla de producción se emplea en las partes derechas de las ecuaciones, el atributo se considerará atributo sintetizado para dicho no terminal.

Recuerde:

- Un atributo no puede ser considerado como heredado y sintetizado al mismo tiempo.

3.3.2 Restricciones contextuales

Para que una gramática en XLOP sea correcta no basta con una correcta especificación sintáctica. La estructura de atributos sintetizados y heredados debe respetar unas restriciones contextuales (de semántica estática) adicionales:

- Un atributo sintetizado perteneciente a un no terminal debe declararse mediante una ecuación en todas las reglas de producción cuya cabeza se corresponda con el no terminal.

- En los atributos de la forma *a of r* o *a of r*(*n*), la referencia sintáctica *r* debe existir en la producción. Además, si el atributo es de la forma *a* of r(*n*), debe existir al menos *n+1* ocurrencias de *r* en dicha producción (esto es porque el número de orden de la primera ocurrencia es 0).

- Los atributos heredados y sintetizados de cada no terminal han de ser disjuntos.

- No tiene sentido asignar más de una ecuación al mismo atributo.

- El único atributo léxico de #pcdata es “text”, a través del cual se obtiene el contenido.

- Sólo se pueden asignar valores a atributos de no terminales en una ecuación.

Aquellas restricciones incumplidas se detectarán como errores y se informará al usuario del presunto error cometido.

Recuerde:

- Para la ejecución de un método de la clase semántica, aunque no devuelva un valor o incluso no queramos almacenarlo, siempre se debe asignar dicho método a un atributo de un no terminal mediante una ecuación. Para estos casos, la asignación se puede realizar a través de un nuevo atributo sintetizado de la cabeza de la producción.

- Las dependencias creadas entre los atributos de la gramática permiten mantener el orden deseado de las operaciones. Estas son lanzadas y ejecutadas lo más pronto posible, cuando ya no dependen de atributos por calcular

3.3.3 La clase semántica

Como ya se ha indicado anteriormente, la clase semántica es una programada en Java que implementa las funciones semánticas definidas en la gramática de atributos XLOP. Cada aplicación XLOP tendrá su propia clase semántica. La clase semántica otorga una completa libertad de operaciones para el usuario a la hora de especificar las operaciones que se realizarán en el procesamiento de un documento XML. En la sección 3.4.4 se muestra un ejemplo de clase semántica.

3.4 Uso de XLOP

El uso de XLOP será detallado con la construcción de un ejemplo paso a paso.

3.4.1 Concepción de la aplicación

Antes de comenzar, se debe tener bien claro el tipo de aplicación de procesamiento de XML que se quiere diseñar. Para ello se debe caracterizar en el tipo de documentos XML a procesar y el procesamiento a realizar.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Desc (Dir | I)*>
<!ATTLIST Desc path CDATA #REQUIRED
              newpath CDATA #REQUIRED>
<!ELEMENT Dir (I)*>
<!ELEMENT I (#PCDATA | is)*>
<!ELEMENT is EMPTY>
```

Figura 3.4.1. Definición de la DTD de la aplicación ejemplo.

Como ejemplo ilustrativo, crearemos una aplicación que permita cifrar o descifrar un mensaje contenido en un archivo de texto. El proceso de cifrado/descifrado se realizará mediante simples cambios de letras o palabras. Estos cambios específicos aparecerán definidos en un archivo XML.

De esta forma, una parte esencial en el desarrollo de una aplicación XLOP es diseñar el lenguaje de marcado específico que será utilizado para marcar los documentos. En nuestro ejemplo, dicho lenguaje viene caracterizado por la DTD de la Figura 3.4.1. Los documentos XML que siguen esta DTD describen un procesamiento de archivos de texto que se realizará de manera secuencial y de principio a fin. Definiremos dos tipos de procesamiento sobre dicho archivo:

1- Lectura secuencial, de principio a fin, realizando la sustitución de una cadena de caracteres por otra. Se realiza mediante la instrucción contenida en un bloque *<I>.*

2- Lectura secuencial, de principio a fin, realizando la sustitución de un carácter por otro, cuyo tipo de reemplazo aparece contenido en una tabla. Se realiza mediante las instrucciones <*I*> contenidas en un bloque <*Dir*>.

Por su parte, los atributos *path* y *newpath* de la etiqueta <Desc> indican, respectivamente, el camino del archivo de texto fuente a codificar, y el camino del archivo resultante de la codificación.

Con todo esto, podemos concretar que los archivos XML en la aplicación de ejemplo estarán compuestos por una serie de instrucciones. Cada instrucción podrá ser del tipo 1 o del tipo 2. Las instrucciones de tipo 1 realizarán una operación básica mientras que las de tipo 2 serán un conjunto de instrucciones básicas. Estas operaciones básicas expresarán el reemplazo de una cadena de caracteres por otra.

Aparte de la importancia que cobra la DTD para caracterizar el formato de los documentos XML objeto de la aplicación, también existen motivos más técnicos que recomiendan disponer de una DTD. El primer motivo se debe a que la DTD aporta validación a los archivos XML. El segundo motivo se debe a que, si no se especifica una DTD, la detección de un posible error pierde precisión y la depuración de una instancia XML resulta costosa. Y el tercer motivo se debe a que, sin la especificación de una DTD, no es posible introducir espacios, tabulaciones o incluso saltos de línea entre las distintas etiquetas de un archivo XML. Sin una DTD, estos espacios, tabulaciones y saltos de línea no son ignorados y se reconocen como #pcdata. Desde un punto de vista técnico, la gramática que define la DTD para las instancias XML debe igual o más general que la definida en nuestra gramática XLOP. Recordemos que la estructura de nuestra gramática estará directamente relacionada con la estructura de etiquetas definida en nuestro archivo XML, y, por tanto, con la DTD.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Desc  SYSTEM "descifrador.dtd">
<Desc path="texto.txt" newpath="textoCifrado.txt">
   <l>egipci<is/>mandarin</l>
   <l>estudio<is/>autopsia</l>
   <l>real<is/>de palo</l>
   <Dir>
      <l>o<is/>a</l>
      <l>a<is/>e</l>
      <l>e<is/>i</l>
      <l>i<is/>o</l>
   </Dir>
   <l>le<is/>mi</l>
</Desc>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Desc  SYSTEM "descifrador.dtd">
<Desc path="textoCifrado.txt" newpath="textoDescifrado.txt">
   <l>mi<is/>le</l>
   <Dir>
      <l>a<is/>o</l>
      <l>e<is/>a</l>
      <l>i<is/>e</l>
      <l>o<is/>i</l>
   </Dir>
   <l>de palo<is/>real</l>
   <l>autopsia<is/>estudio</l>
   <l>mandarin<is/>egipci</l>
</Desc>
```

Figura 3.4.2. Instancias XML para la aplicación de ejemplo.

```
                    ----- Texto -----

El director de la parte egipcia de la expedición, Zahi Hawas, ha anunciado en
una nota del Consejo Supremo de Antigüedades (CSA), el hallazgo de un
cementerio junto al templo de Abusiris en el norte de Egipto, en el que se han
desenterrado 27 tumbas con 10 momias, algunas de las cuales son doradas.

                    ----- Texto cifrado -----

El doricta r di le perti menderone di le ixpidocoón, Zeho Hewes, he enuncoeda
in une nate dil Cansija Suprima di Antogüidedis (CSA), il hellezga di un
cimintiroa junta el timpla di Abusoros in il narti di Egopta, in il qui si hen
disintirreda 27 tumbes can 10 mamoes, elgunes di les cuelis san daredes.
```

Figura 3.4.3. Ejemplo de ejecución de la aplicación Descifrador.

La Figura 3.4.2 muestra dos instancias XML que pueden ser utilizadas en esta aplicación para cifrar y descifrar textos. La instancia XML de la izquierda, produce el cifrado del archivo de texto “texto.txt” en otro archivo de texto “textoCifrado.txt” con las instrucciones definidas en dicho archivo. La Figura 3.4.3 muestra un ejemplo de ejecución en este caso. La instancia XML de la derecha en la Figura 3.4.2, produce el descifrado del archivo de texto “textoCifrado.txt” en otro archivo de texto “textoDescifrado.txt” con las instrucciones que habíamos definido en el primer archivo, pero ahora en orden inverso. Además, para que el cifrado y descifrado se realice correctamente, las instrucciones contenidas en el bloque *Dir* deberán ser permutaciones de caracteres.

3.4.2 Creación de la gramática XLOP

Una vez definida la aplicación que desea desarrollarse, el siguiente paso a realizar consiste en la creación de un archivo de texto .xlop en el que se aloje la gramática de atributos que especifica el procesamiento, definida en el lenguaje de especificación de XLOP. Para ello, es posible utilizar un editor de texto normal y corriente (p.e., el *Bloc de notas* de Windows). La

Figura 3.4.4 ilustra este paso (la gramática es la misma que la de la Figura 3.2.1). El archivo se ha nombrado como ***DesCifrador***.

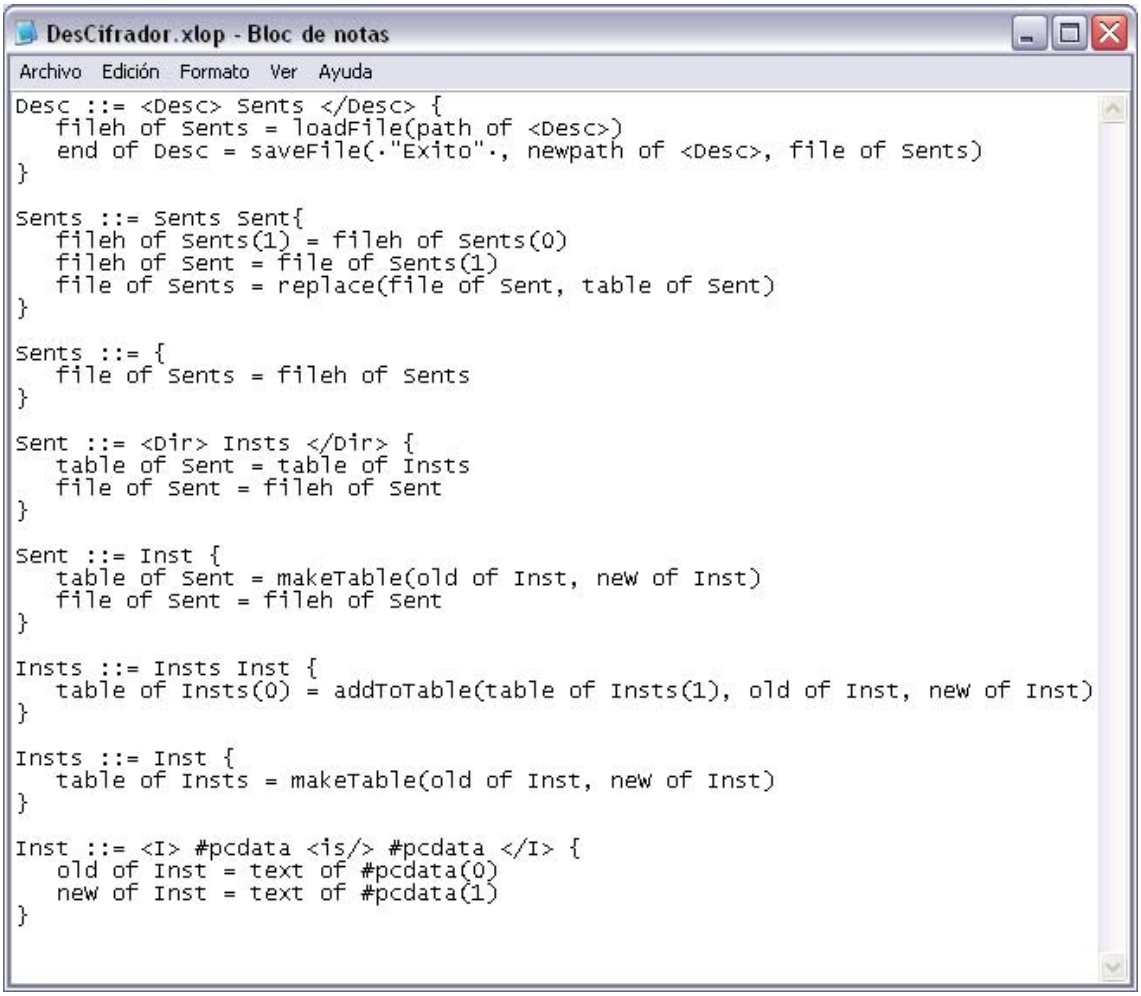

Figura 3.4.4. La gramática XLOP de DesCifrador.

En la gramática XLOP para este ejemplo, *Desc* es el axioma o regla de producción inicial. Por ello, la regla se define en primer lugar a diferencia del resto de reglas de producción. La primera ecuación de esta regla permite cargar el archivo cifrado (el camino estará indicado en el atributo *path*). Nótese que este atributo, propio de una etiqueta, se utiliza en las expresiones semánticas de la producción y existe en la instancia XML, pero sin embargo, no se declara en el cuerpo de la producción.

La carga de la información del fichero de entrada se realiza mediante la función semántica *loadfile(ruta).* Dicha función devuelve la información recopilada en un objeto de la clase File, perteneciente a la *lógica específica de la aplicación* (es decir, el conjunto de clases adicionales que se precisan para llevar a cabo el procesamiento; véase sección 3.4.3). De esta manera, a través de los atributos definidos en la gramática, el descriptor del archivo a descifrar fluirá hacia las ramas más internas del árbol de derivación de la gramática mediante el atributo heredado *fileh*, mientras que en la subida se le aplicarán las operaciones oportunas con los

datos de reemplazo recopilados. Los resultados se guardarán en los atributos sintetizados *file*. Cuando el árbol de derivación es recorrido, la función semántica *saveFile(mensaje, ruta, archivo)* (segunda ecuación de la primera producción), almacena la información procesada en un archivo de texto, en una ruta diferente, y muestra un mensaje.

Las dependencias creadas mediante la propagación de la información a través del árbol de derivación, permite asegurar que el método *saveFile(mensaje, ruta, archivo)* se ejecutará en último lugar y una vez realizadas todas las operaciones sobre la información a cifrar/descifrar.

Las reglas de producción *Sents* otorgan a nuestros archivos XML la capacidad de contener cero o más sentencias. Una sentencia *Sent* aloja un tipo de operación, tipo 1 o 2 (véase Figura 3.4.4). La primera regla *Sent* aloja la operación de tipo 1 basada en una instrucción. La segunda regla *Sent* aloja la operación de tipo 2 basada en un conjunto de instrucciones contenidas en un bloque *Dir* (como estructura de archivo XML).

La primera regla de producción *Sents* realiza, en sus ecuaciones, la operación de tipo 1 o 2 mediante la función semántica *replace(archivo, tabla)*. La función semántica determina el tipo de operación a aplicar según la información almacenada en la tabla. Más concretamente, si la tabla sólo contiene un valor se aplica la operación de tipo 1. En caso contrario, se aplica la operación de tipo 2.

La información que contiene una instrucción *Inst* aparece estructurada mediante una etiqueta de apertura *<I>* y su correspondiente etiqueta de cierre *</I>*. Esto indicará el reemplazo de la cadena de caracteres especificada a la izquierda de una etiqueta vacía *<is/>*, por la cadena de caracteres especificada a la derecha de la etiqueta. Las cadenas de caracteres se obtienen directamente del archivo XML mediante campos de texto XML o #pcdata, y se propagan como atributos sintetizados. Observe que el atributo *neW* aparece con w mayúscula. Esto es debido a que no se permite utilizar palabras reservadas Java en los archivos de especificación XLOP.

En la operación de tipo 2, se construye una tabla debido a que es necesario almacenar la información de las instrucciones en un lugar determinado. Dicha tabla, es un objeto de la clase Table perteneciente a la lógica específica de la aplicación (véase sección 3.4.3). El atributo sintetizado *table* representa nuestra tabla. El llenado de la tabla con información se realiza mediante la función semántica *addToTable(tabla, valor viejo, valor nuevo)* en la primera regla de producción *Insts.* La función semántica requiere una tabla ya construida y la información de una instrucción *Inst*. La creación de la tabla se realiza mediante la función semántica *makeTable(valor viejo, valor nuevo)* en la segunda regla de producción *Insts*. Las dependencias de atributos establecidas en estas reglas, mediante el atributo sintetizado *table,* garantizan el orden de ejecución de las operaciones sucediendo que la tabla será construida antes de ser rellenada.

Recuerde:

- Se puede introducir información de manera directa a los métodos mediante los delimitadores "·". Véase cómo se ha introducido un valor determinado al método *saveFile()* del axioma de la gramática.

3.4.3 Programación de la lógica específica de la aplicación

La información que fluye por el árbol de derivación de la gramática pertenece a instancias de las clases específicas File y Table. Estas clases constituyen la lógica específica de la aplicación y su especificación se realiza en el lenguaje de programación Java.

```
package Logic;

public class File {

   private String data;

   public File() {
      data = new String();
   }

   public void set(String data) {
      this.data = data;
   }

   public String get() {
      return data;
   }
}
```

Figura 3.4.5. Clase File de la apliacación DesCifrador.

La clase File (Figura 3.4.5) permite el almacenamiento y el acceso de información basada en texto, mientras que la clase Table (Figura 3.4.6) permite construir una tabla en donde se asocian pares de valores.

```
package Logic;

import java.util.Hashtable;

public class Table {

   private Hashtable<String,String> table;

   public Table() {
      table = new Hashtable<String,String>();
   }

   public void add(String key, String value) {
      table.put(key,value);
   }

   public String get(String key) {
      return table.get(key);
   }

   public String getNew() {
      return table.values().iterator().next();
   }

   public String getOld() {
      return table.keys().nextElement();
   }
}
```

Figura 3.4.6. Clase Table de la apliacación DesCifrador.

Las aplicaciones realizadas con XLOP pueden depender del uso de librerías, clases java u otro tipo de archivos específicos por parte del usuario.

En nuestro ejemplo, la lectura del archivo de texto se realiza utilizando la clase FileUtils.java (Figura 3.4.7) de la librería Util.jar. Esta clase no es la de la biblioteca estándar de Java, sino que es una clase que ya teníamos implementada y que reutilizaremos para nuestro ejemplo. Dicha clase aporta métodos de gestión de ficheros que facilitan la obtención y almacenamiento de la información en estos formatos. Posteriormente, la importaremos en nuestra clase semántica para poder hacer uso de ella.

```
package Util;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;

public class FileUtil {

   public static String readFile(String path) {
      String resp = "";
      try {
         FileInputStream fis = new FileInputStream(new File(path));
         char buf;
         while ((buf = (char) fis.read()) != '\uffff') {
            resp += buf;
         }
         fis.close();
      } catch (FileNotFoundException e) {
         return "~FileError~";
      } catch (IOException e) {
         return "~FileError~";
      }
      return resp;
   }

   public static String writeFile(String path, String data) {
      String resp = "";
      try {
         Writer output = new BufferedWriter(new FileWriter(new File(path)));
         output.write(data);
         output.close();
      } catch (FileNotFoundException e) {
         return "~FileError~";
      } catch (IOException e) {
         return "~FileError~";
      }
      return resp;
   }
}
```

Figura 3.4.7. Librería Util.FileUtil.

3.4.4 Programación de la clase semántica

La clase semántica es la clase .java que contiene la declaración e implementación de las distintas funciones semánticas utilizadas en la gramática .xlop definida. Cada función semántica se implementa como un método de la clase semántica. Para nuestro ejemplo, es necesario aportar, por tanto, una clase semántica (Figura 3.4.8) que implemente los métodos: *loadfile(ruta), saveFile(mensaje, ruta, archivo), replace(archivo, tabla), addToTable(tabla, valor viejo, valor nuevo), makeTable(valor viejo, valor nuevo).*

```
import Util.FileUtil;
import Logic.*;

public class DesCifrador {

   private boolean error;
   private boolean isChar;

   public DesCifrador() {
      error = false;
   }

   public File loadFile(String path) {
      String text = FileUtil.readFile(path);
      error = text.equals("~FileError~");
      File f = new File();
      f.set(text);
      f.makeTable();
      return f;
   }

   public void saveFile(String msg, String path, File f) {
      if (!error) {
         FileUtil.writeFile(path,f.get());
         System.out.println(msg);
      }
   }

   public File addReplacementPair(String old, String neW, File f) {
      isChar = old.length() == 1;
      f.addTable(old,neW);
      return f;
   }

   public File replacePairs(File f) {
      if (!error) {
         String text = "";
         if (isChar) {
            for (char c:f.get().toCharArray()) {
               if (f.getTable(String.valueOf(c)) != null)
                  text += f.getTable(String.valueOf(c));
               else text += c;
            }
         } else {
            text = f.get().replaceAll(f.getOld(),f.getNew());
         }
         f.set(text);
      }
      f.makeTable();
      return f;
   }
}
```

Figura 3.4.8. Clase Semántica de la apliacación DesCifrador.

Obsérvese que la lógica específica de la aplicación está contenida en la carpeta o paquete Logic (Figura 3.4.5 y Figura 3.4.6), paquete que se importa en la clase semántica. La constructora por defecto de la clase semántica siempre se convierte en el primer método al que se accede al iniciar la ejecución de la aplicación. Por ello, no es necesario realizar llamada a constructora alguna en la gramática XLOP. Sin embargo, es muy importante declarar esta constructora en la clase semántica en el caso de que sea necesario realizar una inicialización de variables del programa.

Nótese que se ha aprovechado la clase semántica para realizar pequeñas comprobaciones de errores a fin de simplificar el ejemplo. Estas comprobaciones podrían haberse también elevado a nivel de especificación de la gramática.

3.4.5 Instalación y ejecución de la aplicación XLOP

La instalación de XLOP sólo requiere descomprimir los archivos en una carpeta. La ejecución de la aplicación se realiza pulsando doble-click sobre el archivo XLOP.jar contenido en la carpeta de instalación. Esto lanzará la interfaz de XLOP de la Figura 3.4.9.

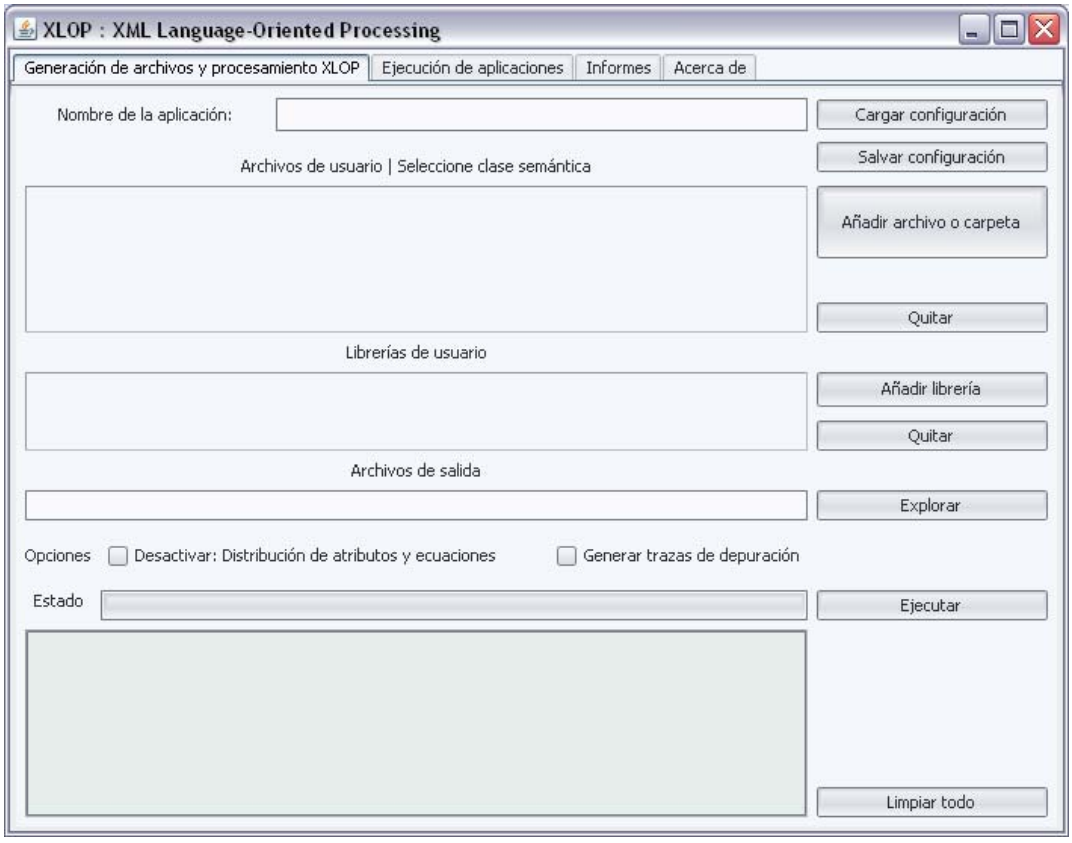

Figura 3.4.9. La interfaz de XLOP.

La interfaz se compone de varias pestañas:

- La pestaña *Generación de archivos y procesamiento XLOP* de la Figura 3.4.9 constituye la pantalla principal de la aplicación. En ella se especifican las rutas de los ficheros del usuario, clase semántica, librerías, la gramática XLOP, y realiza el procesamiento y optimización de la gramática. Por último, genera los archivos que constituyen la aplicación final en la carpeta especificada. También permite *instalar* la aplicación generada, añadiendo todos aquellos archivos adicionales que sean necesarios para su ejecución (p.ej., bibliotecas, DTDs, etc.).

- La pestaña *Ejecución de aplicaciones* es un lanzador de aplicaciones generadas con XLOP. Permite la gestión y ejecución de las aplicaciones de una manera cómoda y sencilla.

- La pestaña *Informes* contiene información sobre el procesamiento y el resultado final de las gramáticas procesadas.

- La pestaña *Acerca de* muestra los autores y la versión de XLOP.

3.4.5.1 La pestaña Generación de archivos y procesamiento XLOP

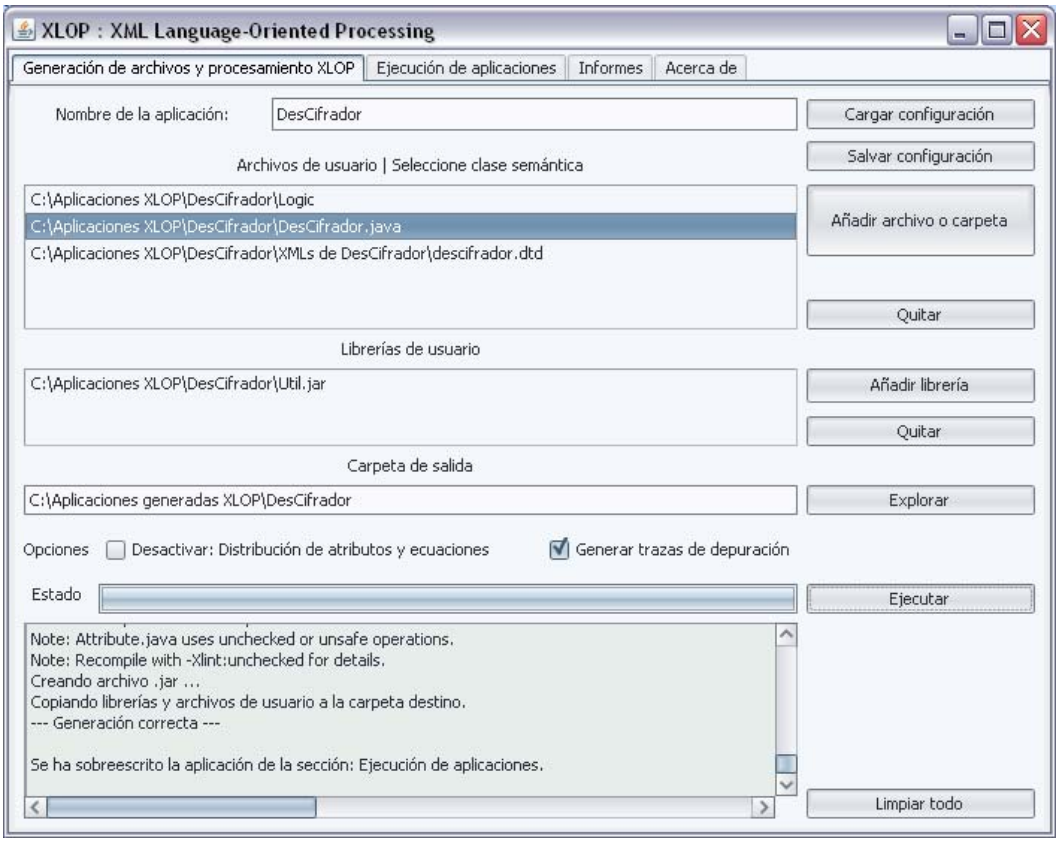

Figura 3.4.10. La pestaña de *Generación de archivos y procesamiento XLOP*.

3.4.5.1.1 Configuración de la aplicación

El botón Ejecutar realiza el proceso de generación de la aplicación, pero previamente, la aplicación ha de configurarse de manera correcta.

- El nombre de la aplicación se introduce en el cuadro de texto *Nombre de la aplicación.* Nombraremos nuestro ejemplo como "DesCifrador".

- A continuación, es necesario determinar la ubicación de los archivos que forman parte de la lógica de específica de la aplicación en la lista ***Archivos de usuario***. Esto se realiza mediante los botones *Añadir archivo o carpeta* y *Quitar.* En esta lista se introducen, principalmente, las ubicaciones de las clases .java, la clase semántica y la DTD de los archivos XML que procesa la aplicación generada. Aquí incluso puede introducirse las rutas de archivos XML concretos que se quieren copiar en la carpeta de destino de la aplicación generada, por si se quiere tener configurado unos primeros casos de uso para la aplicación.

- Añadimos a la lista la clase semántica "DesCifrador.java" y el paquete "Logic", carpeta donde residen las clases File, Table y Descifrador en formato .java.

Es muy importante añadir el archivo .dtd que usan los XML de la aplicación en dicho apartado, sólo en el caso de que dependan de una DTD. Añadimos la DTD "descifrador.dtd". La ubicación del texto que se cifrará o descifrará se ha especificado en los archivos XML mediante una ruta relativa respecto a su directorio de trabajo. El directorio de trabajo es la ubicación donde se almacena la aplicación generada. También debe dejarse seleccionada la clase semántica en la lista anterior.

- En la lista ***Librerías de usuario*** se especifica la ubicación de los archivos .jar, .zip, o .class que requiere nuestra aplicación para su correcta compilación y ejecución. En nuestro ejemplo, es necesario añadir la librería Util.jar, librería encargada de realizar la gestión de ficheros.

- En el cuadro de texto ***Carpeta de salida*** especificamos la carpeta donde se desea guardar la aplicación generada. El botón *Explorar* permite buscar y seleccionar la carpeta de manera más cómoda.

Por último, se nos permite seleccionar varias opciones. Con el marcado de la casilla *Desactivar: Distribución de atributos y ecuaciones,* impediremos que XLOP realice optimización alguna sobre la gramática a procesar. Con el marcado de la casilla *Generar trazas de depuración,* se añaden mensajes informativos durante la ejecución de la aplicación generada sobre cómo se realiza la evaluación de las reglas de producción de la gramática.

Recuerde:

- El directorio de trabajo de nuestra aplicación es la dirección especificada en el cuadro de texto *Carpeta de salida.*
- No olvide dejar seleccionada la clase semántica en la lista de *Archivos de usuario.*

3.4.5.1.2 Guardado de la configuración

Todos los pasos realizados anteriormente se pueden salvar en un archivo de configuración de extensión .pcfg. Los botones *Cargar configuración* y *Salvar configuración* de la esquina superior derecha de la Figura 3.4.10, permiten cargar y guardar respectivamente dichas configuraciones.

3.4.5.1.3 Ejecución de la aplicación

Una vez configurada correctamente la aplicación sólo queda pulsar el botón *Ejecutar* para inicializar el procesamiento. Si todos los pasos anteriores se han realizado correctamente, se nos pedirá la ubicación de la gramática .xlop a procesar. Una vez aceptado el cuadro de diálogo, se inicia el procesamiento.

- La barra de estado muestra el progreso en la generación de nuestra aplicación y debajo se realiza un informe detallado sobre el procesamiento de la gramática y la generación de la aplicación.
- El cuadro de texto *Estado* nos ayudará e informará sobre cualquier tipo de error detectado tanto en la configuración de la aplicación, como cualquier tipo de error detectado en la gramática, e incluso informará sobre errores avanzados.

3.4.5.2 La pestaña Ejecución de aplicaciones

La pestaña que presenta la Figura 3.4.11 se ha diseñado por comodidad y permite ejecutar de manera, simple y directa, todas las aplicaciones XLOP que se generen de manera satisfactoria durante las múltiples ejecuciones de la interfaz.

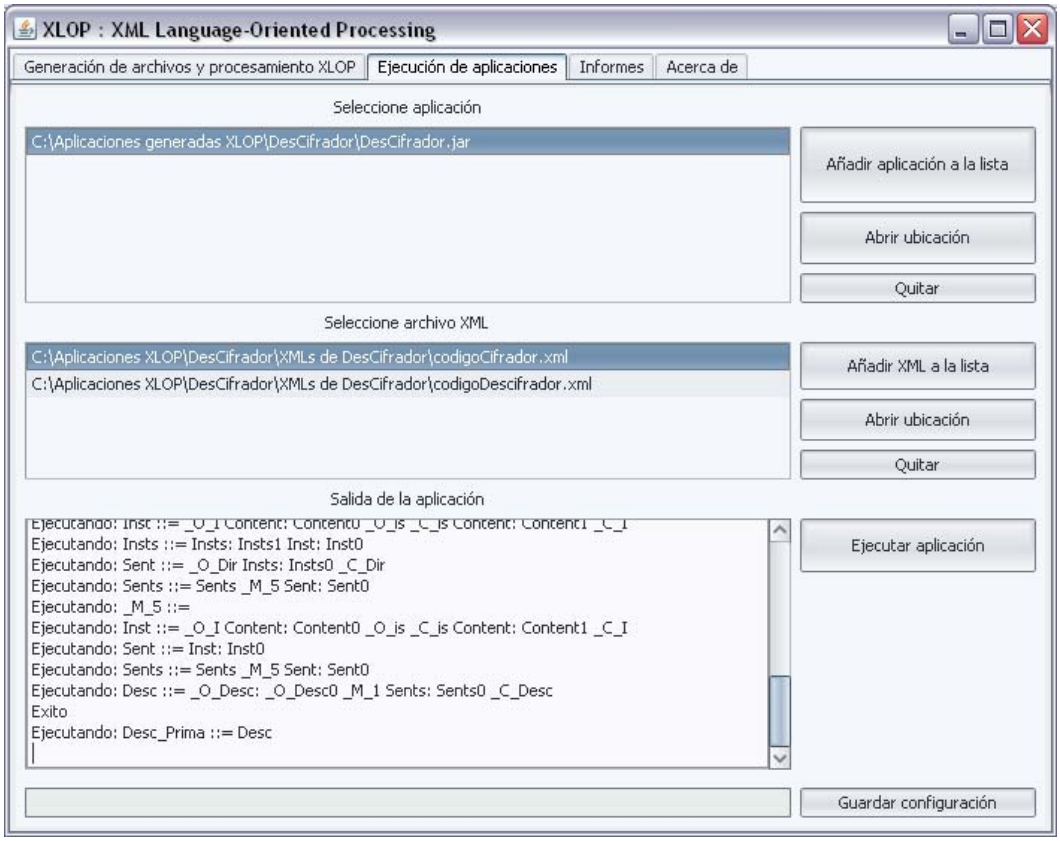

Figura 3.4.11. La pestaña de *Ejecución de aplicaciones*.

3.4.5.2.1 Configuración

La información de esta sección se carga automáticamente desde el archivo "XLOPdata.xdta" ubicado en la capeta de instalación de XLOP. Este archivo no se debe eliminar ni modificar manualmente.

En la lista ***Seleccione aplicación*** aparecerán, de manera automática, las rutas de las aplicaciones generadas con XLOP. Los botones *Añadir aplicación a la lista* y *Quitar* permiten agregar nuevas rutas de aplicaciones o eliminarlas de la lista. El botón *Abrir ubicación* permite abrir la carpeta donde se encuentra la aplicación.

La Figura 3.4.12 muestra el contenido de la carpeta donde reside nuestra aplicación. Puesto que nuestros archivos XML buscarán el texto a cifrar mediante ruta relativa a la aplicación, hemos colocado dicho archivo "texto.txt" en dicha carpeta.

Tras seleccionar una aplicación, se muestra una lista de archivos XML en el cuadro ***Seleccione archivo XML***. Estos son los archivos XML definidos para esa aplicación en concreta, que agregaremos o eliminaremos utilizando los botones *Añadir XML a la lista* y *Quitar*. El botón *Abrir ubicación* permite abrir la carpeta donde se encuentra el archivo XML.

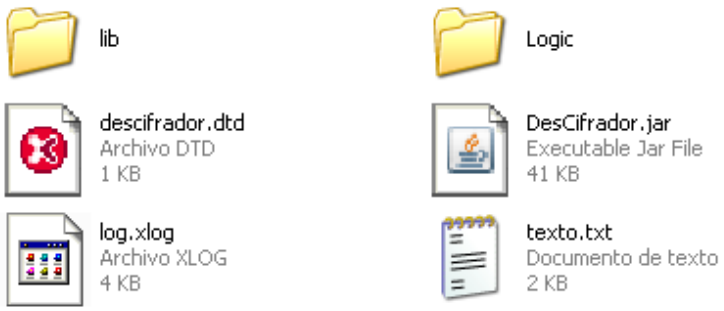

Figura 3.4.12. Contenido de la carpeta de la aplicación DesCifrador.

Recuerde:

- Si la instancia XML depende de diversos y nuevos archivos, es necesario ubicarlos de manera manual en el lugar necesario para su correcta ejecución.

Recuerde:

El concepto de *aplicación* en XLOP está fuertemente *sobrecargado*:

- XLOP en sí es una aplicación que genera aplicaciones.
- Las aplicaciones generadas por XLOP son aplicaciones que procesan documentos XML (es decir, procesadores de lenguajes de marcado específicos).
- En muchas ocasiones, cada documento XML puede entenderse como una descripción de alto nivel de una aplicación (p.ej. cada documento en <e-Tutor> -ver Capítulo 5- *describe* una aplicación: una simulación interactiva en forma de un tutorial interactivo).

Si se consideran los tres niveles, las *aplicaciones* generadas por XLOP son, a su vez, *generadores* de aplicaciones, y la *aplicación* XLOP en sí es un *generador* de *generadores* de aplicaciones [Cleaveland 2001]. Esto está también muy relacionado con los trabajos previos realizados en el grupo <e-UCM> en relación con el *paradigma documental* de construcción de aplicaciones [Sierra et al. 2006; Sierra et al. 2008b].

3.4.5.2.2 Ejecución de la aplicación

Para iniciar la aplicación es necesario dejar seleccionado una aplicación y un archivo XML de las listas anteriores de la Figura 3.4.11.

- El botón *Ejecutar* aplicación inicia la ejecución de la aplicación seleccionada utilizando el archivo XML seleccionado.
- La salida de la aplicación se muestra en el cuadro de texto *Salida de la aplicación*.
- La entrada de la aplicación se gestiona escribiendo en el cuadro de texto inferior.

3.4.5.2.3 Guardado de la configuración

Si se realiza una ejecución y termina de manera correcta, todos los cambios de configuración introducidos se guardan de manera automática. Sin embargo, el botón *Guardar configuración* permite guardar todos los cambios introducidos para que se mantengan presentes en futuras ejecuciones de la interfaz, siempre que se desee.

3.4.5.3 La pestaña Informes

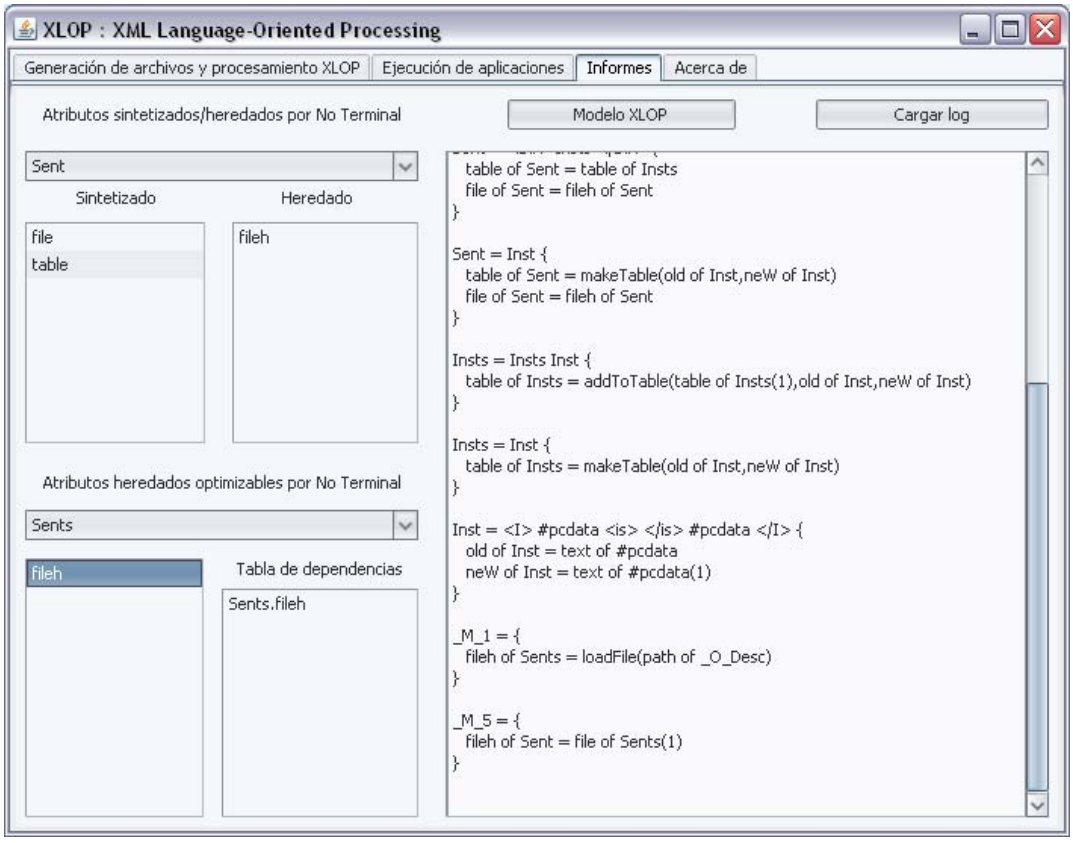

Figura 3.4.13. La pestaña de *Informes*.

La pestaña de la Figura 3.4.13 se muestra un informe detallado de la gramática XLOP que resulta después de realizar las optimizaciones (véase Capítulo 4) y otros datos de interés sobre la última gramática XLOP procesada y válida. Los informes se almacenan automáticamente en un archivo "log.xlog" ubicado en la C*arpeta de salida* correspondiente a cada aplicación generada. Para cargar un log diferente, puede utilizarse el botón *Cargar log.*

Inicialmente, en el cuadro de texto se muestra el resultado de procesar la gramática. Podemos observar la introducción de nuevas producciones _M_ que contienen ecuaciones que se habían especificado en otras producciones. Obsérvese también, cómo estas ecuaciones ya no aparecen en sus producciones originales, al haber sido optimizadas mediante marcadores (véase el Capítulo 4 para los detalles).

El botón *Modelo XLOP* permite mostrar en el cuadro de texto cómo se ha representado la información en el modelo de objetos de XLOP. Esta opción está orientada a prestar información al desarrollador de la herramienta XLOP.

Por último, se nos muestran varias listas con información sobre el carácter adquirido por los atributos de los no terminales que seleccionemos. De esta manera, se puede consultar rápidamente qué atributos de la gramática son sintetizados o heredados. Por su parte:

- La lista de atributos *heredados optimizables* muestra aquellos atributos heredados alojados en marcadores _M_ insertados en la gramática.

- La *Tabla de dependencias* muestra los atributos (a su vez optimizables) de los que depende el atributo optimizable seleccionado.

Capítulo 4
Desarrollo e implementación

4.1 Introducción

El entorno XLOP se compone de módulos específicos que realizan el tratamiento de las gramáticas definidas en el lenguaje de especificación de XLOP a fin de producir automáticamente aplicaciones de procesamiento de documentos XML. En dicho tratamiento se presta especial atención a la generación de aplicaciones optimizadas, que permitan reducir el consumo de memoria y el tiempo de ejecución que requiere el procesamiento de los documentos XML.

En este capítulo se describe el desarrollo de XLOP. Se comienza introduciendo la metodología de desarrollo adoptada. Seguidamente, se presenta la arquitectura del entorno XLOP, describiendo los pasos más importantes que se realizan en la generación de una aplicación XLOP. A continuación, se detallan los principales componentes y uso de los marcos / herramientas JavaCC, SAX y CUP, así como se desarrollan los detalles de implementación de los algoritmos principales encargados del tratamiento y optimización de la gramática XLOP aportada por el usuario.

4.2 Método de desarrollo

Una de las principales dificultades de este proyecto se ha debido a que, al situarse el desarrollo en el contexto de un proyecto de investigación, el alcance de los objetivos y los algoritmos no han estado claros desde un primer momento. Esta situación de incertidumbre añade importantes dificultades al desarrollo. Efectivamente, a medida que se ha avanzado en el desarrollo del proyecto, han surgido casos no triviales que, al no haberse tenido anteriormente en cuenta, han implicado cambios tanto a nivel algorítmico como a nivel estructural. Por otra parte, dichos algoritmos han requerido conocimientos muy puntuales y concretos sobre propiedades específicas que se derivan de los lenguajes basados en gramáticas de atributos, tanto para la correcta implementación de los algoritmos, como a la hora de validar los resultados de las pruebas. El hecho de que el desarrollo de los módulos siguientes dependa del funcionamiento de los módulos anteriores, implica que cada nueva iteración y proceso añadido impacta incrementalmente en la dificultad, tanto para el almacenamiento y obtención de nueva información, como para el nuevo procesamiento de dicha información, como para la verificación de la funcionalidad del nuevo módulo y verificación del proceso completo. Esto es debido a que el proceso realizado por cada módulo depende directamente (e inevitablemente) de todas las transformaciones previas, al seguir el generador de XLOP un proceso necesariamente secuencial. Y por otra parte, estos errores

generados han tenido que ser estudiados meticulosamente, al poderse tratar en cada caso, de fallos de especificación de algoritmos o fallos de implementación.

De esta forma, con el fin de gestionar adecuadamente las dificultades surgidas en este contexto tan cambiante, se ha seguido una metodología de desarrollo ágil [Larman 2003] basada en iteraciones cortas, con reuniones semanales con el equipo científico del proyecto (en particular, con los profesores José Luis Sierra y Antonio Sarasa), en las que se han establecido los objetivos de cada siguiente iteración, y se ha analizado el trabajo realizado, los problemas surgidos durante la iteración previa, y los pasos a llevar a cabo para resolver los citados problemas.

4.3 Arquitectura del Sistema

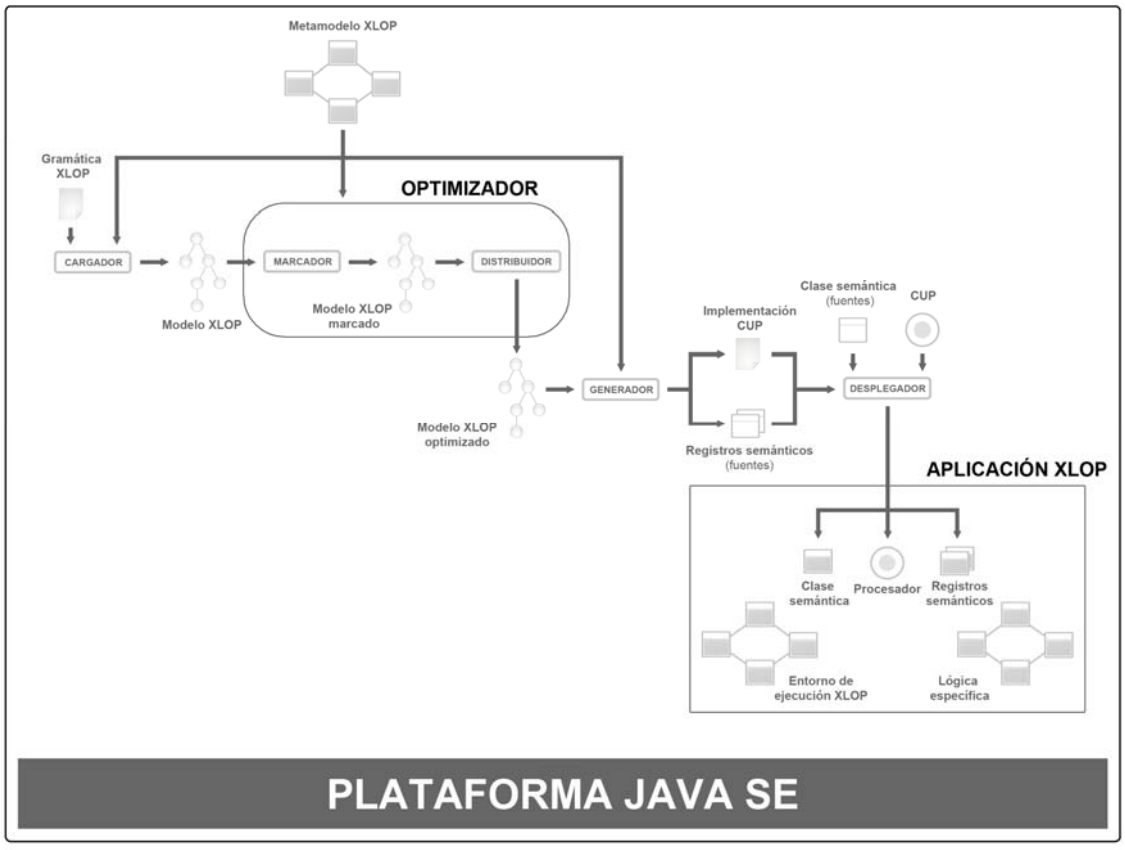

Figura 4.3.1. La arquitectura del entorno XLOP.

La Figura 4.3.1 muestra la arquitectura del entorno XLOP. Dicho entorno funciona enteramente sobre la plataforma Java SE. Los principales módulos del entorno se coordinan en torno a una representación abstracta de las gramáticas XLOP, como árboles de objetos Java. Esta representación está normada por el metamodelo XLOP: un conjunto de clases Java que caracteriza la sintaxis abstracta de XLOP. El módulo cargador realiza el análisis sintáctico-semántico de las gramáticas XLOP y construye un modelo XLOP normado por el metamodelo

XLOP. Antes de generar dicho modelo, el módulo cargador verificará la estructura de atributos de la gramática XLOP a fin de buscar incoherencias, generando un informe de errores detallado, si ello procede. El módulo optimizador se encarga de realizar una optimización sobre la gramática original a partir del modelo XLOP, a fin de generar una representación que conduzca a una implementación rápida y eficiente. El módulo optimizador está compuesto, a su vez, por dos submódulos:

- El módulo marcador: Analiza la estructura de la gramática original a través del modelo XLOP, y genera una gramática marcada sobre dicho modelo, produciendo un modelo XLOP marcado.

- El módulo distribuidor: Analiza las relaciones y dependencias que aparecen en los atributos de la gramática marcada a través del modelo XLOP marcado, y genera una representación optimizada, produciendo el modelo XLOP optimizado.

El módulo Generador es un módulo que genera una implementación en código CUP, a partir de un modelo XLOP optimizado. Junto al código CUP, genera los registros semánticos asociados a los no terminales y marcadores utilizados para gestionar los atributos de los mismos. El módulo desplegador obtiene la aplicación a partir de la traducción del código CUP a Java, y la compilación de la clase semántica y de los registros semánticos. La aplicación en sí se completa con el entorno de ejecución XLOP (invariante para todas las aplicaciones) y la lógica específica de la aplicación.

En resumen, los componentes aportados por el usuario para la construcción del núcleo de la aplicación que procesa documentos XML deberán ser:

- La gramática en el lenguaje de especificación XLOP, como entrada al módulo cargador. Esto es, un archivo de especificación .xlop.

- La Clase Semántica que implementa los métodos contemplados en la gramática XLOP, como entrada para el módulo Desplegador.

- La lógica específica de la aplicación. Clases o librerías que se precisan para llevar a cabo satisfactoriamente el procesamiento de los documentos.

Las siguientes secciones están dedicadas a concretar el funcionamiento de cada uno de los módulos principales que constituyen la arquitectura del entorno XLOP.

4.4 El metamodelo XLOP

En Ingeniería de Lenguajes Software y desarrollo de software dirigido por modelos, un *metamodelo* es un modelo de un lenguaje [Kepple 2008]. De esta forma, el metamodelo XLOP es una caracterización abstracta de las gramáticas describibles en el lenguaje de especificación de XLOP. Es un modelo de clases Java diseñado pensando en un acceso y almacenamiento

simple, estructurado, completo y sin redundancia de la información. Sus instancias se denominan modelos XLOP, y cada una de ellas representa una gramática XLOP.

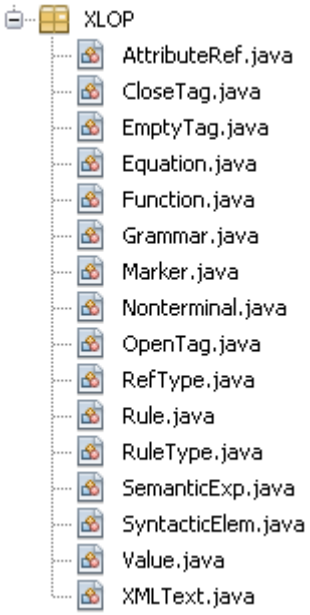

Figura 4.4.1. Desplegado de clases del metamodelo XLOP.

La Figura 4.4.1 presenta las diversas clases que forman parte del metamodelo XLOP. Nótese que aparece una clase adicional EmptyTag (representa una etiqueta vacía) que, aunque completamente funcional, se ha descartado su uso en la actual versión del entorno XLOP.

4.4.1 Estructura del metamodelo

La Figura 4.4.2 muestra la estructura del metamodelo XLOP mediante un diagrama de clases.

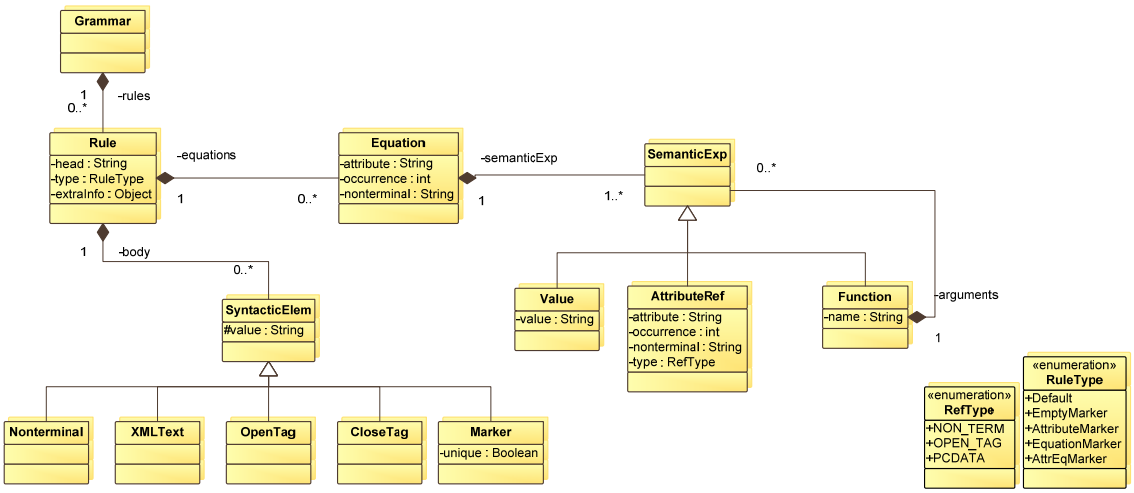

Figura 4.4.2. El metamodelo XLOP. Diagrama de clases.

Los nodos raíz de los modelos XLOP vienen caracterizados por la clase Grammar. Una gramática está compuesta por cero o más reglas. Estas reglas son instancias de la clase Rule. Una regla está representada por la cabeza y los elementos que constituyen el cuerpo de la producción. La cabeza de una regla se almacena en el atributo ·head de la clase Rule. Los elementos que constituyen el cuerpo de la producción son instancias de la clase abstracta SyntacticElem.

Los elementos del cuerpo de la producción contienen un atributo ·value heredado de la clase SyntacticElem. ·value almacena el nombre del elemento extraído de la gramática procesada. Estos elementos son los siguientes:

- Nonterminal: representa un no terminal.
- XMLText: representa un texto XML. Por defecto el atributo ·value toma el valor "#pcdata".
- OpenTag: representa una etiqueta de apertura.
- CloseTag: representa una etiqueta de cierre.
- Marker: representa un marcador. ·value está formado por un número con prefijo "_M_". ·unique indica que no existe un marcador igual en el cuerpo de otra regla de producción cualquiera.
- Value: representa un valor no tipado.

Cada una de estas instancias se almacenan en el atributo ·body de la clase Rule en el mismo orden en el que aparecen en la gramática.

Adicionalmente, cada regla contiene cero o más ecuaciones semánticas. Cada una de estas ecuaciones se almacena en el atributo ·equations de la clase Rule y, al igual que sucede con el atributo ·body, se almacena en el mismo orden en el que aparecen en la gramática. Estas ecuaciones son instancias de la clase Equation.

Una ecuación queda definida por una parte izquierda y una parte derecha: intuitivamente, "el valor de la parte derecha se asigna al elemento de la parte izquierda". La parte izquierda de una ecuación únicamente podrá estar formada por una referencia a atributo de un no terminal, referencia que queda representada mediante los siguientes atributos:

- ·attribute: nombre del atributo del no terminal.
- ·occurrence: número de ocurrencia del no terminal.
- ·nonterminal: nombre del no terminal.

La parte derecha de una ecuación contiene al menos una expresión semántica, clase abstracta SemanticExp. Estas instancias se almacenan en el atributo ·semanticExp en el mismo

orden en el que aparecen presentes en la gramática. Una expresión semántica puede ser de dos tipos: una referencia a atributo o una función.

Las referencias a atributos se representan mediante la clase AttributeRef. La información que almacenan estas referencias es la siguiente:

- ·attribute: nombre del atributo del no terminal.
- ·occurrence: número de ocurrencia del no terminal.
- ·nonterminal: nombre del no terminal.
- .type: permite indicar el tipo de elemento al que refiere dicho atributo. Estos tipos son tres: no terminal, etiqueta de apertura y texto XML.

Las funciones se representan mediante la clase Function. El atributo ·name almacena el nombre de la función. El número de argumentos de una función varía entre cero y un número determinado, pero cada argumento está formado sólo por uno de los siguientes dos tipos: una función o una referencia a atributo. De esta forma, una función queda completamente representada cuando los argumentos se almacenan en el atributo ·arguments en el mismo orden en el que aparecen en la gramática.

Por último, mencionar que la clase Rule posee dos atributos adicionales (·type y ·extraInfo) que se detallarán y utilizarán más adelante, al igual que la clase Marker.

4.4.2 Recorrido y manipulación de los modelos XLOP

Los modelos XLOP se recorren mediante el uso de un patrón de diseño especial: el patrón Visitor [Gamma et al. 1994]. Para ofrecer dicho comportamiento, las clases del metamodelo XLOP implementan un método apropiado y requerido por el diseño del patrón. Éste es el método *accept(Visitor visitor)*. De la misma manera, se ha definido una interfaz Visitor cuya implementación permite dicho recorrido aprovechando las ventajas que ofrece el patrón.

La Figura 4.4.3 presenta la estructura genérica del patrón Visitor (véase [Gamma et al. 1994]). En concreto, para el metamodelo XLOP, la interfaz Visitor se compone de trece métodos *visit*() diferentes (un método por clase específica del metamodelo XLOP). Por otra parte, todas las clases del metamodelo XLOP poseen la implementación del método *accept(Visitor visitor)* de la manera que muestra la Figura 4.4.3.

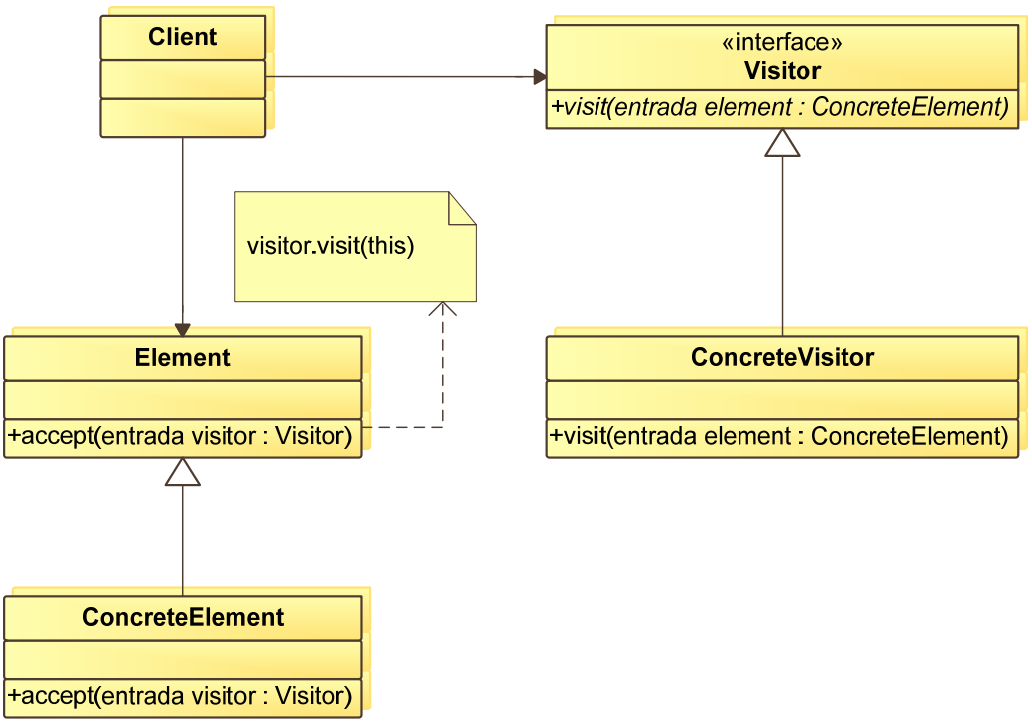

Figura 4.4.3. El patron Visitor.

A continuación, a modo de introducción y referencia, se indican las distintas clases concretas que implementan la interfaz Visitor:

- La clase DoVisitorCE: Comprueba si se cumplen las restricciones contextuales (sección 4.5.3) sobre la gramática, a partir de la información aportada por el modelo XLOP y la tabla de símbolos (sección 4.5.2). Ámbito de uso: módulo Cargador.

- La clase DoVisitorLALR: Traduce las reglas de la gramática XLOP del modelo XLOP en reglas simples requeridas por el módulo Marcador. Actualiza el modelo XLOP con marcadores una vez que se haya finalizado el proceso de marcado de la gramática. Ámbito de uso: módulo Marcador.

- La clase DoVisitorDA: Verifica los elementos contenidos en una lista de atributos considerados como optimizables y los actualiza. No realiza modificaciones sobre el modelo XLOP. Ámbito de uso: módulo Optimizador.

- La clase DoVisitorDA_E: Realiza la optimización del modelo XLOP (sección 4.7.1) sobre el modelo XLOP marcado, obteniendo una gramática optimizada. Ámbito de uso: módulo Optimizador.

- La clase DoVisitorCUP: Genera el código perteneciente al archivo parser.cup y de los registros semánticos (sección 4.8.1.2) para la construcción del núcleo de la aplicación final. Ámbito de uso: módulo Generador.

Cada una de las clases mencionadas implementa todos los métodos de la clase Visitor. Cada método se corresponde con la operación que se realiza sobre un tipo de objeto determinado. La ejecución del método específico asociado a cada tipo de objeto se realiza mediante la operación *visit(objeto a visitar),* sin necesidad de tener que conocer el tipo de objeto a evaluar.

Recuerde:

- No todas las clases visitantes que implementan la interfaz Visitor, implementan de manera significativa a su vez, todos sus métodos. Sin embargo, tampoco se realizan llamadas a estos métodos no implementados.

4.5 El Cargador

El Cargador es el módulo que permite realizar el análisis de una gramática definida en el lenguaje de especificación de XLOP y generar un modelo XLOP que aloje, de manera estructurada y accesible, toda la información contenida en la gramática de partida.

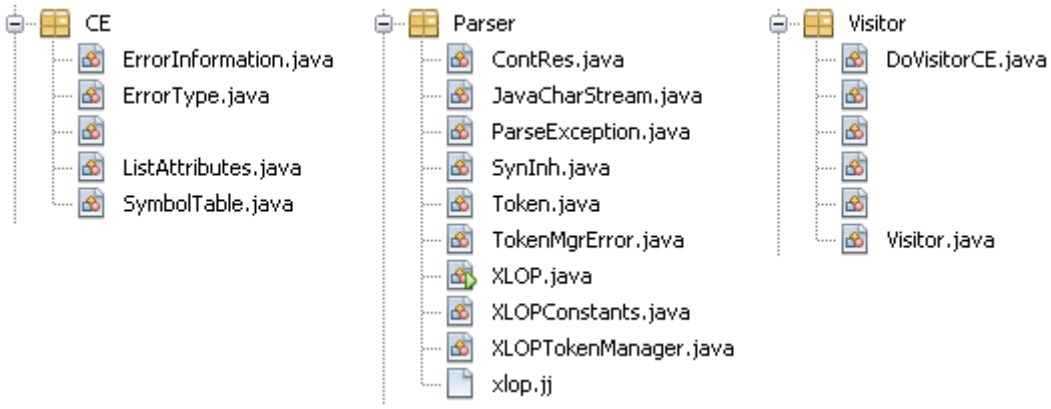

Figura 4.5.1. Desplegado de clases del módulo Cargador.

La Figura 4.5.1 presenta las diversas clases que intervienen en el proceso que realiza el módulo Cargador.

4.5.1 Construcción del cargador con JavaCC

El módulo Cargador es, en realidad, un traductor de gramáticas XLOP a modelos XLOP. Se ha implementado utilizando como herramienta el generador de analizadores sintácticos JavaCC (véase sección 2.3.2). El archivo xlop.jj contiene la implementación del analizador sintáctico XLOP en el lenguaje de programación especifico y requerido por JavaCC.

Siguiendo la presentación realizada en la sección 2.3.2, el contenido del archivo se divide en bloques de la siguiente manera:

- Declaración de métodos y variables estáticas: en este bloque se declaran los métodos estáticos y las variables estáticas a las que se accederá en las diferentes etapas del análisis sintáctico. El contenido de este bloque se especifica en el lenguaje de programación Java. Para la ejecución del cargador es necesario crear un método estático en esta sección que llame al método parser (ruta del fichero fuente) que ejecutará las reglas de producción especificadas según el contenido del un fichero de entrada.

- Declaración de Tokens y caracteres especiales: en este bloque se especifican los caracteres que van a definir de forma léxica cada palabra y se les asigna un nombre identificador a cada palabra. Cada vez que se lea una secuencia de caracteres y coincida con una especificada, se reconocerá por el nombre que se ha asignado y se devolverá un *Token*. Un Token almacena información sobre la palabra reconocida; lexema, nombre y posición en el archivo entre otros. El orden en el que se declaran los Tokens es importante. El orden determina el tipo de Token que se asociará a la secuencia de caracteres leída, basándose en el primero que encaje. La declaración de los Tokens se realiza en el bloque TOKEN. Se han declarado dos bloques TOKEN, uno para las definiciones léxicas y otro para las definiciones auxiliares (véase Figura 4.5.2). Además de una mayor claridad, asegura el orden correcto de ejecución en el reconocimiento de los Tokens.

```
 ...

TOKEN :  /* Definicion Lexica */
{
   < Of: "of" > |
   < LlaveAp: "{"> |
   < LlaveC: "}"> |
   < Texto_XML: "#pcdata" > |
   < Definicion: "::=" > |
   < Igual: "=" > |
   < Parentesis_Abierto: "(" > |
   < Parentesis_Cierre: ")" > |
   < Numero: "0" | ["1"-"9"](["0"-"9"])* > |
   < Coma: "," > |
   < IdentificadorCaracteresJava: <JavaLetra> (<JavaLetra> | <JavaDigito>)* >|
   < IdentificadorXML: (<LetraXML> | "_" | ":") (<CaracteresNombreXML>)* > |
   < Etiqueta_Apertura: "<" <IdentificadorXML> (<SXML>)? ">" > |
   < Etiqueta_Cierre: "<" "/" <IdentificadorXML> (<SXML>)? ">" > |
   < Etiqueta_Vacia: "<" <IdentificadorXML> (<SXML>)? "/" ">" >
}

TOKEN :  /* Definiciones Auxiliares */
{
   < #SXML: ("\u0020" | "\u0009" | "#xD" | "#xA")+ > |
   < #CaracteresNombreXML: <LetraXML> | <DigitoXML> | "." | "-" | "_" | ":" |
                           <CaracteresCombinadosXML> | <ExtendidosXML> > |
   < #CaracteresCombinadosXML:
      ["\u0300"-"\u0345", "\u0360"-"\u0361", "\u0483"-"\u0486", "\u0591"-
"\u05A1",
       "\u05A3"-"\u05B9", "\u05BB"-"\u05BD", "\u05BF", "\u05C1"-"\u05C2",
"\u05C4",

 ...
```

Figura 4.5.2. Contenido del archivo xlop.jj. Sección de declaración de Tokens.

- Declaración de reglas de producción: en este bloque se declaran las reglas de producción de la gramática en el lenguaje propio de JavaCC y se especifican las operaciones que se realizarán en cada punto en el lenguaje de programación Java. En esta sección se realizan las operaciones más importantes, como son la creación de la Tabla de Símbolos (sección 4.5.2). la comprobación de las restricciones contextuales del lenguaje de especificación XLOP (sección 4.5.3) y la creación del modelo de objetos XLOP (sección 4.5.4).

A la hora de transformar las reglas a la sintaxis de JavaCC, se puede observar un parecido en la forma de declarar métodos en la sintaxis de Java, tal y como muestra la Figura 4.5.3.

```
...

void Lista_Elementos_Sintacticos(Rule rg): {}
{
   Elemento_Sintactico(rg) Resto_Lista_Elementos_Sintacticos(rg)
}

void Resto_Lista_Elementos_Sintacticos(Rule rg): {}
{
   Elemento_Sintactico(rg) Resto_Lista_Elementos_Sintacticos(rg) | {}
}

void Elemento_Sintactico(Rule rg): {Token tok = null;}
{
   Elemento_XML(rg) | (tok = No_Terminal() | tok = <Texto_XML>)
   {
      if (tok.kind == Texto_XML) {
         rg.addElement_Sintax(new XMLText(tok.image));
      } else {
         rg.addElement_Sintax(new Nonterminal(tok.image));
      }
      /* Comprobar restricciones contextuales */
      ntpAct = contRes.addlProd(tok.image, ntpAct);
   }
}
 ...
```

Figura 4.5.3. Contenido del archivo xlop.jj. Sección de declaración de reglas de producción.

En el fragmento de código de la Figura 4.5.3, cuando se aplica la regla *Elemento_Sintactico*, se construye el objeto correspondiente a un no terminal o un texto XML según el Token leído. Por último, se guarda el Token en un objeto de la clase contRes para posteriormente comprobar las restricciones contextuales.

Tras compilar el archivo xlop.jj con JavaCC se genera una implementación del Cargador constituida por los archivos Java mostrados en la Figura 4.5.4.

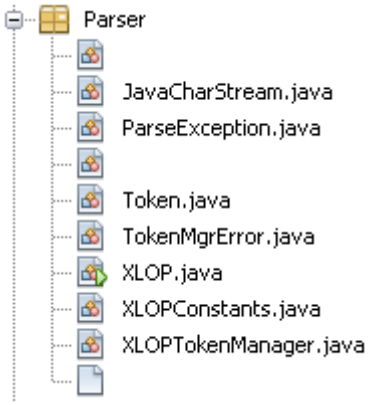

Figura 4.5.4. Clases generadas por JavaCC mediante la compilación del archivo xlop.jj.

4.5.2 La tabla de símbolos

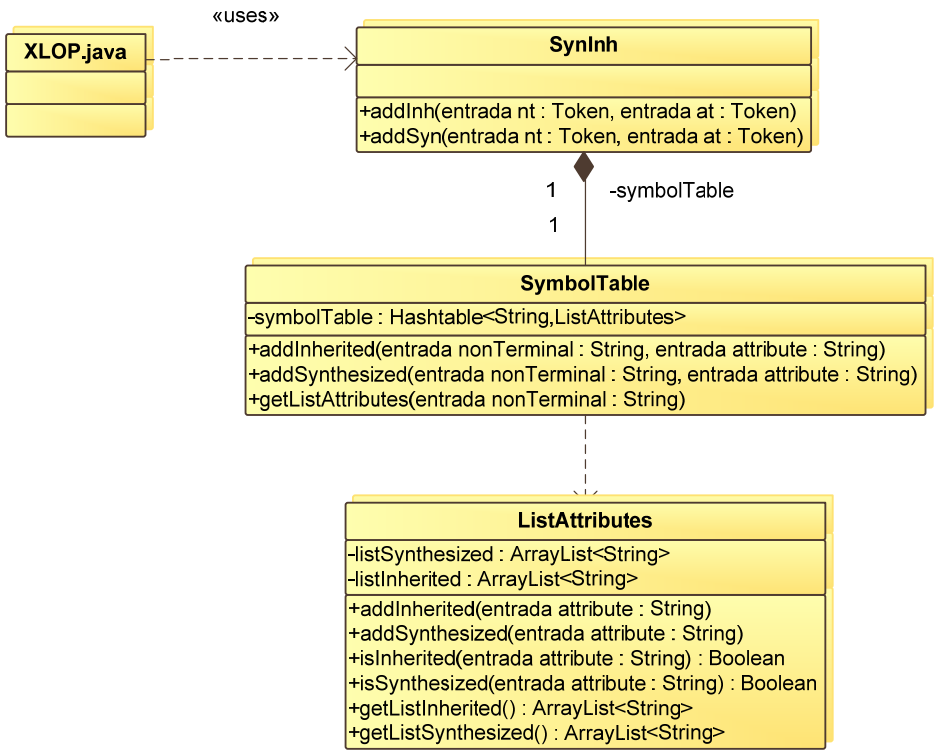

Figura 4.5.5. Diagrama de clases de gestión de la tabla de símbolos.

El diagrama de clases de la Figura 4.5.5 resume las principales clases involucradas en el manejo de la tabla de símbolos. De esta forma, la tabla de símbolos es una tabla que asocia a no terminales, dos conjuntos: el conjunto de atributos sintetizados y el conjunto de atributos heredados, para dicho no terminal. Durante el procesamiento realizado por el módulo Cargador, cada atributo de no terminal detectado se deberá clasificar como heredado o sintetizado, mediante su correcta inserción en la tabla de símbolos.

Para contener toda la información clasificada por atributos heredados/sintetizados para cada atributo de cada no terminal, se dispone de la clase SynInh. Esta clase sirve para contener de manera estructurada la información y permitir el acceso y creación de la tabla de símbolos de manera sencilla. La tabla de símbolos es un objeto de la clase SymbolTable, cuya estructura es la misma que la definida al comienzo de la sección. Adicionalmente, la clase SymInh almacena las incoherencias y errores producidos por una estructura de atributos incorrecta. Estos errores son los siguientes:

- "El atributo "[atributo]" de "[no terminal]" no puede ser sintetizado puesto que ya consta como heredado."
- "El atributo "[atributo]" de "[no terminal]" no puede ser heredado puesto que ya consta como sintetizado."

Los atributos que producen conflicto no se almacenan en la tabla de símbolos. En cambio, se almacena el error que han producido.

4.5.3 Comprobación de restricciones contextuales

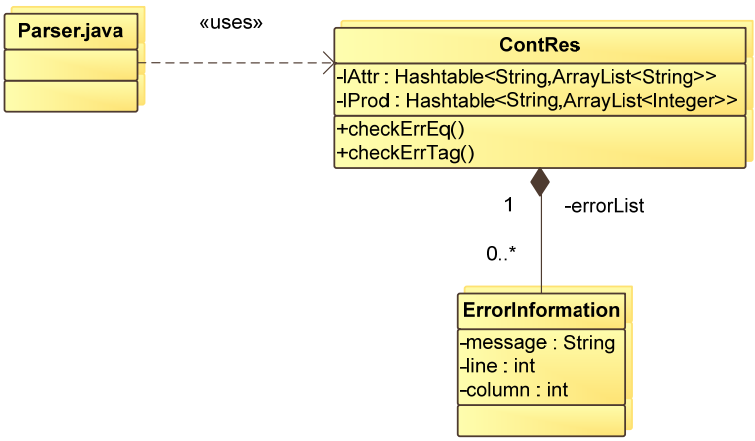

Figura 4.5.6. Diagrama de clases de comprobación de restricciones contextuales.

La Figura 4.5.6 muestra las clases utilizadas para la comprobación de las restricciones contextuales. La comprobación de las restricciones contextuales se realiza de manera sencilla mediante la clase ContRes. Esta clase almacena los errores encontrados en una lista de objetos ErrorInformation, objetos que almacenan información detallada sobre cada error. La clase contRes consta de dos tablas:

- ·lAttr: Almacena como clave la cadena formada por el "número de ocurrencia + nombre" del no terminal, y como valor, su lista de atributos. Esto permite comprobar que no existan varias ecuaciones asignadas a un mismo atributo en una misma ecuación.

- ·lProd: Almacena como clave el nombre del no terminal, y como valor las posiciones del no terminal en la regla. Esto permite comprobar que los números de ocurrencia especificados en las ecuaciones semánticas sean válidos, y permite detectar producciones inexistentes.

Las comprobaciones anteriores se realizan mediante el método *checkErrEq(),* método que se aplica a cada referencia atributo dentro de las declaraciones semánticas de una regla de producción del archivo xlop.jj (de la Figura 4.5.3). Los errores que detecta el método son los siguientes:

- “No tiene sentido realizar una asignacion al atributo de un #pcdata o de una etiqueta”.
- “El único atributo permitido de #pcdata es: text".
- "No existe la producción: "[cabeza de producción].
- "El numero de ocurrencia ("[número]") de "[no terminal]" no existe. (Solo existen de 0 a "[valor máx. de número de ocurrencia para la regla]")".
- "No pueden existir dos ecuaciones distintas para el mismo atributo en una misma producción: "[atributo]" of "[no terminal].

Por otra parte, existe un método *checkErrTag*() que comprueba si se declararon con el mismo nombre, la etiquete de apertura y su etiqueta de cierre asociada. El error se detecta como:

- "La etiqueta de cierre " [nombre] " no coincide con la de apertura " [nombre]

Todos los errores llevan asociados el punto exacto donde se detectó el error, indicando el número de línea y columna del archivo fuente. Hay que añadir que esta clase consta de dos métodos newlAttr() y newlProd() que vacían las tablas ·lAttr y ·lProd. Esto es debido a que las comprobaciones sólo se realizan a nivel de regla de producción.

4.5.4 Creación del modelo XLOP

El modelo de objetos XLOP que se deriva como instancia del metamodelo XLOP, se genera durante el proceso de análisis secuencialmente a la lectura del fichero que contiene la gramática XLOP aportada por el usuario. Las reglas definidas en el archivo de implementación del parser xlop.jj realizan la creación de cada una de las instancias del metamodelo XLOP y, su inserción en el mismo orden en que aparecen especificados dichos componentes en la gramática. De esta manera se consigue representar la gramática XLOP contenida en un archivo fuente mediante una representación arbolescente de objetos Java con la misma exactitud. Por último, es importante indicar que, siguiendo el convenio de XML, las etiquetas vacías

especificadas en la gramática XLOP se representan como dos objetos en el modelo XLOP, formado por la etiqueta de apertura y su correspondiente etiqueta de cierre.

4.6 El Marcador

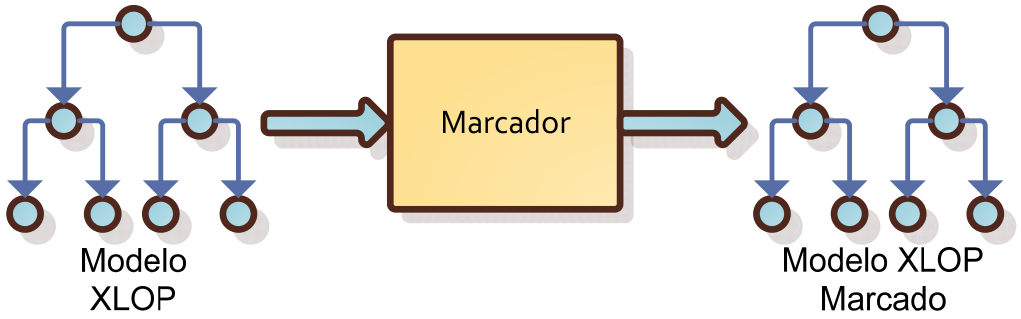

Figura 4.6.1. Módulo marcador. Genera un modelo XLOP marcado a partir del modelo XLOP .

El módulo marcador mejora el modelo XLOP insertando marcadores en dicho modelo para obtener el modelo XLOP marcado (Figura 4.6.1). El módulo se basa en una adaptación de los algoritmos descritos en [Purdom&Brown 1980].

Efectivamente, a partir del modelo generador por el cargador el objetivo es realizar el cálculo de las posiciones de la gramática en la que se pueden insertar marcadores, nuevos no terminales definidos mediante reglas vacías, de manera que la gramática original no pierda el carácter que poseen las gramáticas LALR(1). El módulo entonces procede a insertar dichos marcadores en el modelo XLOP para obtener el modelo XLOP marcado. No todos los marcadores insertados son utilizados en la gramática que se genera en la última etapa de optimización, pero dichos marcadores nos informan de las posibles posiciones de la gramática en la que se pueden alojar atributos heredados, así como instrucciones que permiten adelantar la ejecución de las ecuaciones semánticas. Almacenar atributos heredados en marcadores, y adelantar la evaluación de las instancias de atributos significa evitar la espera del cálculo de atributos y reducir el consumo de memoria.

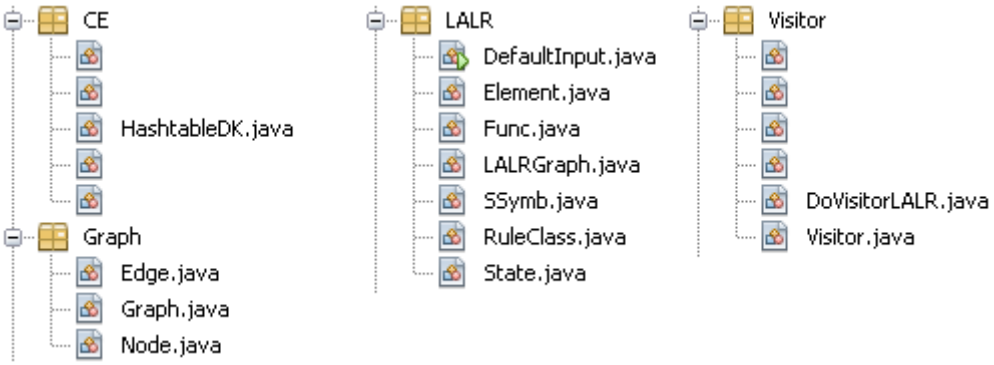

Figura 4.6.2. Desplegado de clases del módulo Marcador.

La Figura 4.6.2 presenta las diversas clases que intervienen en el proceso que realiza el módulo Marcador. Nótese que existe una clase ejecutable DefaultInput que permite probar el funcionamiento del módulo Marcador de manera independiente, al ser un módulo altamente reutilizable cuyo funcionamiento no depende de una representación concreta de la gramática a procesar. Además, nótese que existe una clase HashtableDK que implementa una tabla hash con la finalidad de facilitar el acceso e inserción de un conjunto de valores asociados a una misma entrada.

4.6.1 El autómata LALR

El autómata LALR para una gramática es el motor de un analizador ascendente por desplazamiento-reducción para dicha gramática [Aho et al. 2007]. Como todos los autómatas finitos que gobiernan este tipo de analizadores, dicho autómata reconoce el lenguaje de los prefijos viables de la gramática, es decir, las posibles formas sentenciales que aparece en la pila de análisis del analizador durante el reconocimiento. Así mismo, como todos estos autómatas finitos, sus estados contienen información para decidir cuándo realizar las *reducciones*: substituir el cuerpo de una producción en la cima de la pila (el *asidero*) por la cabeza de dicha producción. La ventaja del método LALR es que, con un número de estados razonable -igual al del método LR(0)-, tienen unas capacidades muy buenas de detectar los asideros, evitando, de este modo, posibles *conflictos.* Un conflicto se produce cuando es posible, bien *desplazar* un terminal y al tiempo reducir por una regla, o bien reducir por dos reglas diferentes.

Los estados del autómata LALR(1) se corresponden con conjuntos de *elementos* LALR(1): conceptualmente, un par de la forma $[A \rightarrow \alpha.\beta,\Phi]$, donde $A \rightarrow \alpha.\beta$ es una producción, y Φ es un conjunto de terminales denominados *símbolos de preanálisis*. La idea es que, si, ante una configuración de la pila, el autómata se encuentra en un estado en el que hay un elemento de la forma $[A \rightarrow \gamma.,\Phi]$, y el siguiente símbolo en la entrada es un terminal *a* que está en Φ, entonces es posible reducir por la regla $A \rightarrow \gamma$. Esto no evita necesariamente los conflictos, ya que en ese mismo estado puede haber otro elemento de la forma $[B \rightarrow \alpha.a\beta,\Omega]$ (lo que permitiría también *desplazar a* a la cima de la pila de análisis), o bien $[B \rightarrow \eta.,\Omega]$, con *a* en Ω, lo que permitiría también *reducir* por $B \rightarrow \eta$. No obstante, para una amplia clase de gramáticas (las gramáticas LALR(1)), el método *funciona* (véase [Aho et al. 2007] para más detalle)

¿Cuáles son exactamente los estados de un autómata LALR(1)? Conceptualmente, la manera más fácil de *caracterizarlos* es [Grune&Jacobs 2008]:

- Partir de un autómata finito *no determinista* reconocedor de prefijos viables LR(1). Los estados de este autómata son *elementos* LR(1) de la forma $[A \rightarrow \alpha.\beta,a]$, con *a* un terminal. Las transiciones son de dos tipos: $[A \rightarrow \alpha.X\beta,a]=X=>[A \rightarrow \alpha X.\beta,a]$, y $[A \rightarrow \alpha.B\beta,a]=\lambda=>[B \rightarrow .\gamma,b]$, para cada terminal b en los *primeros* de βa (un terminal *b* es un primero de una cadena α cuando de α se puede derivar una cadena que comienza por *b*; formalmente, $\alpha =>^* b\alpha'$).

- Aplicar a este autómata la *construcción por subconjuntos* [Aho et al. 2007] para obtener un autómata finito determinista equivalente: el autómata LR(1). En este autómata, cada estado estará etiquetado por conjuntos de elementos LR(1). Si tomamos, de estos elementos, las primeras componentes (es decir, los $A \rightarrow \alpha.\beta$), obtenemos lo que se llama *el corazón* del estado.

- El autómata LALR(1) se obtiene, entonces, *fusionando* los estados del autómata LR(1) que tienen el mismo corazón. Los símbolos de preanálisis de cada elemento LALR(1) integrarán los segundos componentes de los correspondientes elementos LR(1) que lo han originado.

Como resumen, en cara a la implementación, para realizar el proceso de marcado de una gramática es necesario disponer de toda la información que aporta el autómata finito determinista LALR(1) asociado a la gramática. Dicho autómata se representa por un grafo etiquetado y dirigido. Cada nodo del grafo, representa un estado; y cada estado, se compone de un conjunto no vacío de elementos. Estos elementos no son más que reglas de producción de la gramática original sobre las que se han marcando una posición específica del cuerpo de la producción, junto con un conjunto de símbolos de preanálisis para dicha posición.

4.6.1.1 El paquete LALR

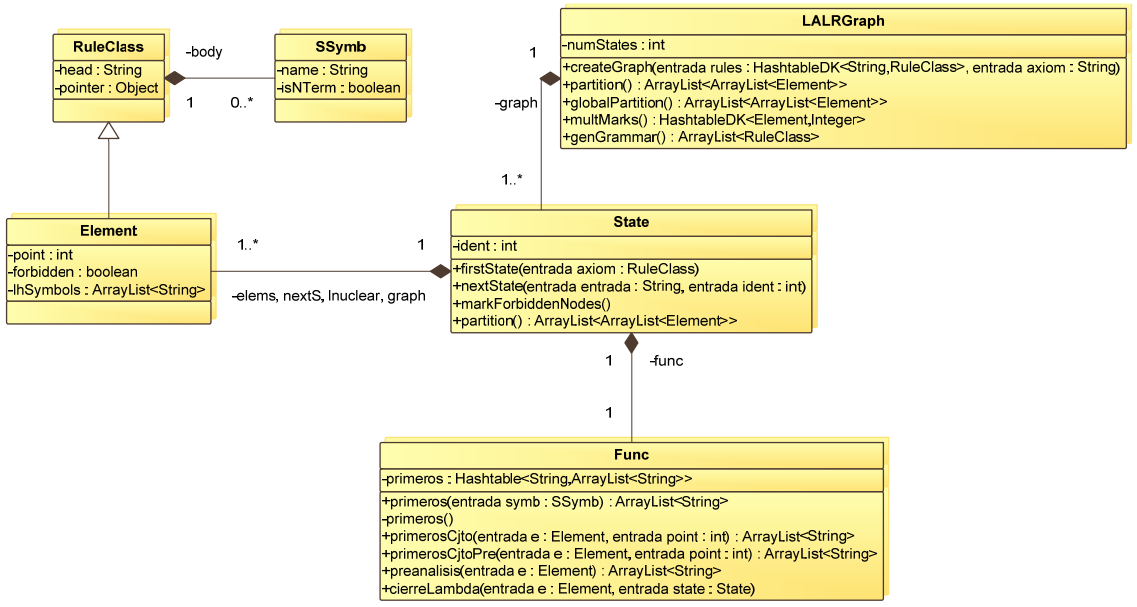

Figura 4.6.3. Relaciones entre las clases participantes en el proceso de marcado de gramáticas.

La estructura de clases utilizada para realizar la representación del autómata LALR(1) está contenida en el paquete LALR, cuyas principales clases poseen todas las operaciones necesarias para la construcción y generación automática del autómata finito determinista LALR(1) correspondiente a la gramática original. La construcción de dicho autómata es necesaria en el proceso de cálculo de marcadores. El diagrama de clases de la Figura 4.6.3

representa las relaciones y dependencias que presentan las principales clases y métodos que intervienen en dicha representación. Cada una de las clases y operaciones se detallan con más precisión en cada uno de los próximos apartados.

4.6.1.2 Reglas gramaticales y elementos

El generador del autómata LALR utiliza estructuras de datos especialmente diseñadas para agilizar el proceso. Esta implementación es independiente a la representación del metamodelo XLOP. Al ser sencilla y genérica, es fácil y rápida tanto de construir como de recorrer, y lo más importante, permite que el algoritmo sea altamente reutilizable.

De esta manera, cada regla de producción de una gramática se convierte previamente en una instancia de la clase RuleClass. Nótese que a este nivel de cálculo no tiene sentido utilizar información sobre las referencias a atributos y ecuaciones de una gramática. Este conjunto de reglas que representa la gramática a procesar se almacena en una estructura especial. La estructura se basa en una tabla cuyas entradas permiten múltiples valores. Esta tabla se implementa mediante la clase auxiliar HashtableDK. La clave de la tabla consiste en la cabeza de una regla de producción y dicha entrada almacena todas las reglas que contienen la misma cabeza. Sólo existe una instancia de la gramática transformada a este conjunto de reglas, la cuál es utilizada por las clases LALRGraph, State y Func únicamente con carácter de consulta. El atributo contenedor de la gramática simplificada se identifica en todas las clases como ·rules.

En la Figura 4.6.3 se puede observar que la clase Element hereda de RuleClass. Una instancia de la clase RuleClass representa una regla de producción de la gramática. La cabeza de la regla *head* se almacena en un String debido a que siempre representará un no terminal, mientras que los elementos del cuerpo se almacenan de forma secuencial en una lista como objetos NTerm, cuyo orden de inserción determina su orden de aparición en el cuerpo de la regla. Un objeto SSymb determina si el elemento es un no terminal, o un terminal, almacenando el nombre de dicho elemento. Además, cada instancia de la clase RuleClass posee un campo ·pointer que permite almacenar un objeto cualquiera. Este campo se utiliza para guardar la regla del modelo XLOP, es decir, de dónde se ha derivado.

Una instancia de la clase Element representa una regla de producción en donde una posición del cuerpo de la regla está marcada y además, almacena los símbolos de preanálisis correspondientes a esa posición. También indica si la posición que marca es una posición establecida como prohibida (véase sección 4.6.3). En concreto, para una instancia de la clase Element se cumple:

- La clase Element es una extensión de la clase RuleClass.
- La posición marcada ·point se determina mediante un número entero.
- Los símbolos de preanálisis ·lhSymbols se representan mediante una lista de String.

- El booleano ·forbidden indica si la posición es prohibida.

Debido a que por cada regla RuleClass existirá posteriormente múltiples elementos Element, el contenido de la instancia RuleClass se comparte para todos los elementos Element. La información de cada instancia RuleClass no se modificará salvo cuando ya no se utilicen los elementos Element. De esta manera se evita la duplicación innecesaria de información.

4.6.1.3 Los estados

Los estados del autómata LALR se representan mediante la clase State. Un estado del autómata LALR se define por un conjunto de elementos Element. La clase State posee una serie de operaciones que permiten generar tanto el primer estado del que parte el autómata, como el siguiente estado de transición a partir de una entrada especificada. Las principales componentes de un estado son:

- El conjunto de elementos Element**.** Se implementa mediante la tabla hash especial HashtableDK<String,Element> ·elems. La clave de la tabla es la cabeza del elemento y como valor contiene el conjunto de elementos con dicha cabeza. De esta manera se puede acceder en tiempo medio constante a todos los elementos de una determinada cabeza de producción.

- La tabla HashtableDK<String,Element> ·nextS. Contiene los mismos elementos de la tabla ·elems, pero estructurada de manera diferente. En este caso, una clave de la tabla es el terminal o no terminal colocado en la posición marcada del elemento Element que almacena. ·nextS sirve para obtener, en tiempo constante, el conjunto de reglas de producción inicial que componen el siguiente estado.

- Un identificador numérico único contenido en el campo ·ident. Esto permite ordenar y diferenciar estados entre sí de manera simple y directa.

El contenido de las tablas ·elems y ·nextS no está duplicado, puesto que sólo almacenan punteros a elementos.

4.6.1.3.1 Creación del primer estado

El estado inicial del autómata LALR se compone de un conjunto de elementos derivados del axioma o regla de producción inicial de la gramática, incluyéndola. Sin embargo, este axioma no es exactamente el axioma de la gramática original, sino un nuevo axioma cuyo único elemento del cuerpo consiste en el axioma original. El primer elemento del autómata se genera a partir del axioma estableciendo {$} como conjunto de símbolos de preanálisis. La posición del punto en el cuerpo del elemento se establece delante del primer terminal/no terminal. La creación del primer estado se realiza mediante la creación de una instancia de la

clase State y mediante la ejecución del método *firstState(axioma)*. Este método permite crear el primer estado del autómata a partir de la regla de producción inicial de la gramática.

El proceso de generación del primer estado consiste en el siguiente:

- Se crea un elemento Element a partir de la regla de producción inicial obtenida como una instancia de la clase RuleClass. A este elemento se le asocia como conjunto de símbolos de preanálisis {$} y se marca dicho elemento como *posición nuclear* (elemento que no puede generarse dentro de un mismo estado mediante la aplicación de la función cierre-lambda), agregándose a la lista ·lnuclear.
- Los siguientes elementos se calculan mediante la aplicación de la función *cierreLambda(elemento,* estado*)* sobre dicho elemento. Dicha función también genera un subgrafo que representa cómo se relacionan los elementos según la manera en que se han derivado, subgrafo que se almacena en el atributo ·graph.

La función *cierreLambda(elemento, estado)* se detalla en la sección 4.6.2.2.

4.6.1.3.2 Generación del siguiente estado

La generación del siguiente estado se realiza aplicando una entrada de transición a un estado. La clase State posee un método *nextState(entrada transición, numero identificador nuevo estado)* que permite la creación del siguiente estado a partir del actual, dada una entrada de transición válida. La entrada de transición se compone de un terminal o de un no terminal, y dicha entrada será válida si el estado actual posee algún elemento Element en cuya posición marcada refiera a dicho terminal o no terminal. Este conjunto de elementos se obtiene de manera directa mediante el acceso a la tabla ·nextS dado el elemento anterior como clave. A partir de este primer conjunto de elementos obtenidos de la tabla, se calcula un nuevo conjunto de elementos que difieren únicamente en la posición marcada, aumentada en una unidad. Este conjunto será candidato a ser las *posiciones nucleares* e iniciales del siguiente estado, junto a los símbolos de preanálisis que aportan. Los restantes elementos se calculan mediante la aplicación de la función *cierreLambda(elemento,estado)* sobre el conjunto de elementos calculado. Recalcar de nuevo que las posiciones nucleares son los nodos que no se han obtenido mediante un cierre-lambda en el grafo considerado del estado.

4.6.2 Algoritmos

La clase Func implementa las funciones auxiliares necesarias en la generación de un estado. Estos algoritmos son:

- Cálculo de los símbolos primeros de un terminal/no terminal.
- Cálculo de los símbolos primeros de un conjunto.
- Cálculo de los símbolos de preanálisis a partir de un elemento.

- La función cierre-lambda.

A continuación se detallan dichos algoritmos.

4.6.2.1 Cálculo de los símbolos primeros

El cálculo de símbolos primeros se implementa de acuerdo al algoritmo de la Figura 4.6.4. Este algoritmo es implementado mediante el método privado *primeros*() de la clase Func, método que sólo se ejecuta una única vez y de manera automática al crear la primera instancia de dicha clase. El algoritmo es una adaptación directa de los descritos en [Grune&Jacobs 2008]. De esta forma, los primeros de cada no terminal se almacenan en una tabla. Por su parte, el método *primeros(s)* permite obtener el conjunto de primeros de cualquier símbolo gramatical (sea éste un terminal, o sea éste un no terminal; ver Figura 4.6.5).

```
primeros() {
   Para cada no terminal A {
      ·primeros[A] := listaVacía()
   }
   boolean Modificado
   Repetir {
      Modificado := falso
      Para cada no terminal A {
         Para cada producción A->X1X2…Xn {
            Tratado := falso
            i = 1
            Mientras ¬Tratado y i≤n {
               Si Xi es un terminal {
                  Si ·primeros[A] no contiene Xi {
                     Añadir Xi a ·primeros[A]
                     Modificado := cierto
                  }
                  Tratado := cierto
               } si no { //Caso no terminal
                  Tratado := cierto
                  Para cada símbolo de ·primeros[Xi] {
                     Si símbolo ≠ λ {
                        Si Primeros[A] no contiene símbolo {
                           Añadir símbolo a ·primeros[A]
                           Modificado := cierto
                        }
                     } si no {
                        Tratado := falso
                     }
                  }
               }
               i := i+1
            }
            Si ¬Tratado {
               Añadir λ a ·primeros[A]
               Modificado := cierto
            }
         }
      }
   } mientras Modificado
}
```

Figura 4.6.4. Algoritmo de cálculo de símbolos primeros para un no terminal.

```
ListaSímbolos primeros(symb: SSymb) {
   Si symb es un terminal {
      Devolver crearLista(symb.name)
   } si no {
      Devolver ·primeros[symb.name]
   }
}
```

Figura 4.6.5. Algoritmo de cálculo de símbolos primeros de un terminal/no terminal.

4.6.2.1.1 Cálculo de conjuntos de primeros de una posición

Una operación recurrente del optimizador es calcular el conjunto de primeros de β en una posición $A \rightarrow \alpha.\beta$. Para ello, se define una operación *primerosCjto*(*r*,*p*), que, dada una regla *r* y una posición *p* en el cuerpo de dicha regla, calcula el conjunto de primeros de la forma sentencial que queda a la derecha de dicha posición. La Figura 4.6.6 describe dicho cálculo.

```
ListaDeSímbolos primerosCjto(A->X1X2…Xn: RuleClass, posición: int) {
   símbolos := listaVacía()
   Repetir {
      símbolosTmp := primeros(Xposición)
      Para cada símbolo de símbolosTmp {
         Si símbolos no contiene a símbolo y símbolo ≠ lambda {
            Añadir símbolo a símbolos
         }
      }
      Si símbolosTmp no contiene a lambda {
         posición := n+1
      }
      posición = posición + 1
   } Mientras posición ≤ n
   Si posición = n+1 {
      Añadir lambda a símbolos
   }
   Devolver símbolos
}
```

Figura 4.6.6. Algoritmo de cálculo de *conjuntos de primeros* de una posición.

4.6.2.1.2 Cálculo de los símbolos de preanálisis a partir de un elemento

El algoritmo de cálculo de los símbolos de preanálisis se implementa de acuerdo al algoritmo de la Figura 4.6.7. Los símbolos de preanálisis se calculan en el método *preanalisis(elemento)*. Sin embargo, este método no realiza más que una llamada a un método más general, *primerosCjtoPre(elemento, posición)*, utilizando la posición marcada más una unidad como argumento. Gracias a los algoritmos anteriores, el método se simplifica drásticamente.

```
ListaDeSímbolos preanalisis(elemento: Element) {
   Devolver primerosCjtoPre(elemento, elemento.damePosicionMarcada() + 1)
}

ListaDeSímbolos primerosCjtoPre([A->X1X2…Xn,Ω]: Element, posición: int) {
   Si posición = n+1 {
      Devolver Ω de A
   }
   símbolos := primerosCjto(A, posición)
   Si símbolos contiene lambda {
      Eliminar lambda de símbolos
      símbolos := unión(símbolos, Ω)
   }
   Devolver símbolos
}

Nota: Ω son los símbolos de preanálisis ya asociados al elemento A
```

Figura 4.6.7. Algoritmos de cálculo de símbolos de preanálisis de una cadena en base al algoritmo de cálculo de símbolos primeros.

4.6.2.2 Cálculo del cierre-lambda de los elementos

La función cierre lambda implementa el algoritmo de la Figura 4.6.8. El método que lo implementa construye a su vez un grafo que hace explícita dicha relación entre los elementos. Estos grafos serán esenciales en la determinación de los marcadores. El método se invoca para cada posición nuclear de cada estado.

```
cierreLambda(elemento: Element, estado: State) {
   grafo := grafoDe(estado)
   Añadir elemento a grafo
   Apila elemento en pendientes
   Mientras pendientes no sea vacía {
      elem := desapila(pendientes)
      Si elem es de la forma [A->α.Bβ,Ω] con B un no terminal {
         Γ := preanalisis(elem)
         Para cada regla B->γ {
            elem' := [B->.γ,Γ]
            Si ya existe elem' en grafo {
               Si Ω ∩ Γ ≠ vacío {
                  Γ := unión(Ω, Γ)
                  Apilar elem' en pendientes
               } si no hay un arco de elem a elem' {
                  nuevaAristaDirigidaDeHacia(grafo,B,elem,elem')
               }
            } si no {
               Apilar elem' en pendientes
               Añadir elem' a grafo
               nuevaAristaDirigidaDeHacia(grafo,B,elem,elem')
            }
         }
      }
   }
}
```

Figura 4.6.8. Implementación de la función cierre-lambda. Creación/modificación del subgrafo asociado al estado.

4.6.2.3 Construcción y revisión del autómata

```
CreaGrafo(reglas, axioma) {
   grafo := grafoVacío()
   estado := nuevoEstado(reglas,0//ID)
   estado := estado.firstState(axioma)
   Añadir estado a grafo
   listaEstados := {estado}
   listaCambiados := listaVacía()
   estadoExistente := vacío

   Para cada estado de listaEstados {
      listaTransición := listaEntradasTransición(estado)
      Para cada entrada de listaTransición {
         estadoNuevo := estado.nextState(entrada,ID(último(listaEstados))+1)
         Si listaEstados contiene a estadoNuevo {
            estadoExistente := buscarEstadoIgualA(listaEstados,estadoNuevo)
            Si algún símbolo de preanálisis de algún elemento de
             estadoExistente difiere de estadoNuevo {
               unir símbolos de preanálisis de elementos nucleares de
                estadoExistente con estadoNuevo
               Recalcular cierre-lambda de los elementos nucleares de
                estadoExistente
               Añadir estadoExistente a listaCambiados
            }
         }
         Si estadoExistente es vacío {
            Añadir nuevoEstado a grafo
            Añadir nuevoEstado a listaEstados
            nuevaAristaDirigidaDeHacia(grafo,entrada,estado,estadoNuevo)
         } si no {
            nuevaAristaDirigidaDeHacia(grafo,entrada,estado,estadoExistente)
         }
      }
   }

   // Recalcular símbolos de preanálisis de estados cambiados
   Para cada estado de listaCambiados {
      listaTransición := listaEntradasTransición(estado)
      Para cada entrada de listaTransición {
         estadoNuevo := estado.nextState(entrada,-1//parámetro no relevante)
         estadoExistente := buscarEstadoIgualA(listaEstados,estadoNuevo)
         Si algún símbolo de preanálisis de algún elemento de estadoExistente
             difiere de estadoNuevo {
            unir símbolos de preanálisis de elementos nucleares de
             estadoExistente con estadoNuevo
            Recalcular cierre-lambda de los elementos nucleares de
             estadoExistente
            AñadirÚltimo estadoExistente a listaCambiados
         }
      }
   }
}
```

Figura 4.6.9. Algoritmo de generación del autómata LALR.

La clase LALRGraph representa el autómata determinista LALR de la gramática. Su construcción se realiza a partir del conjunto de reglas en formato simplificado, reglas

RuleClass, que representan la gramática de partida. El autómata se representa mediante un grafo cuyos nodos son instancias de la clase State. El grafo se almacena en el atributo ·graph de la clase.

La Figura 4.6.9 presenta el método *createGraph(reglas, axioma)* que permite la construcción de dicho autómata.

4.6.3 Detección de Posiciones Prohibidas

Los siguientes procesos permiten determinar las *posiciones prohibidas* de cada estado. Una posición prohibida es una posición del cuerpo de una producción sobre la que no es posible alojar un marcador sin que se produzca un conflicto en la gramática resultante. Las posiciones prohibidas son elementos Element cuya posición determinada por el punto indica el lugar exacto donde no es posible colocar un marcador.

Debe indicarse que la determinación de los criterios para la detección de las posiciones prohibidas ha sido, y continua siendo, un tema activo de investigación en el proyecto Santander/UCM PR34/07-15865. De esta forma, se ha optado por estructurar esta fase como un encademiento de procesos, a fin de facilitar la adición de nuevas etapas. De hecho, una vez cerrada la fase de desarrollo de este proyecto de Sistemas Informáticos, se ha descubierto una nueva etapa que no se incluye en el trabajo entregado. Dicha etapa solventa algunas situaciones patológicas. Sin embargo, es necesaria para garantizar la completitud del algoritmo. Desde este punto de vista, el algoritmo presentado en esta memoria debe considerarse como un algoritmo heurístico (incompleto). Así mismo, una vez que se converja a una solución completa, será posible realizar una refactorización de la cadena de etapas, a fin de obtener una solución más eficiente.

4.6.3.1 Detección de Ciclos

La primera fase en la detección de las posiciones prohibidas se realiza mediante el algoritmo de detección de ciclos. Efectivamente, es posible demostrar que si se marca una posición asociada con un elemento en un ciclo, la gramática resultante presenta un conflicto.

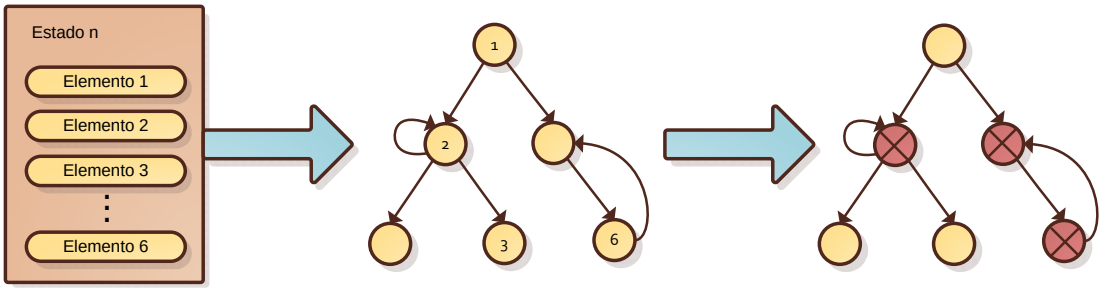

Figura 4.6.10. Proceso de detección de ciclos.

Para cada estado, se obtiene de sus elementos el grafo que explicita la forma según la cuál se han generado, es decir, la forma en que se transita de un elemento a otro mediante la aplicación de la función cierre-lambda (véase sección 4.6.2.2). De esta forma, se pueden obtener varios conjuntos de nodos sin ninguna conexión entre sí, puesto que los nodos raíces siempre estarán formados por las *posiciones nucleares*. Las posiciones nucleares (véase sección 4.6.1.3) constituyen los nodos raíces de los subgrafos. Los ciclos se producen cuando existe un camino en el grafo cuyo origen y destino coinciden en un mismo nodo. El nodo origen/destino y todos los nodos incluidos en dicho camino se consideran nodos prohibidores. Estos nodos se corresponden con un elemento concreto del estado. Por ello, las posiciones que aparecen en estos elementos se marcan como posiciones prohibidas.

```
Ciclos(V: Lista_Vértices,g: Lista_Adyacencia){

   Prohibidos[] := Vector_Bool(V.size());
   Visitados[] := Vector_Bool(V.size());
   Prohibidos[] := InicializarA(falso);
   /*Dado un vértice se investigan todos los caminos que pueden salir de dicho
     vértice*/
   Para cada v de V {
      /*En la pila guardo los vértices que voy visitando en el orden de
        visita*/
      Pila_Vacia(pila);
      Apila(pila,v);
      Visitados[] := InicializarA(falso);
      /*En visitados indico si un vértice ya ha sido visitado en un recorrido
        desde la raiz*/
      Marcar(visitados,v);
      /*Comienzo la búsqueda de ciclos*/
      Buscar-Ciclos(v,g)
      DesMarcar(visitados,v);
   }
}

Buscar-Ciclos(v:Vértice, g:Lista_Adyacencia){
   /*Se van considerando los distintos vértices adyacentes a uno dado*/
   Para cada s:Vértice adyacente a v {
      Si ¬Buscar(visitados,s){
      /*El siguiente vértice a estudiar aún no ha sido visitado. No hay
        ciclo*/
         Apila(pila,s);
         Marcar(visitados,s);
         Buscar_Ciclos(s,g);
         DesMarcar(visitados,s);
         Desapila(pila,s);
      } si no {
      /*El siguiente vértice a estudiar aún ha sido visitado. Hay ciclo.
        Se recorre la pila hasta encontrar el comienzo del ciclo. Todos los
        vértices comprendidos, entre los extremos del vértice se marcan como
        prohibidos*/
         Marcar_Prohibidos(Prohibidos,pila,s);
      }
   }
}
```

Figura 4.6.11. Algoritmo de detección de ciclos para un grafo implementado mediante listas de adyacencia.

La detección de ciclos sigue el algoritmo de la Figura 4.6.11, que está presente en el paquete o librería Graph utilizada en el almacenamiento y representación de la información de grafos.

4.6.3.2 Prohibición por niveles

La prohibición por niveles es un algoritmo que se aplica a continuación de la detección de ciclos. Permite detectar posiciones prohibidas adicionales que no detecta el primer algoritmo. Efectivamente, es posible demostrar que si en un mismo nivel de profundidad hay dos elementos unidos por un camino, todas las posiciones del camino (excepto la del último elemento) son prohibidas.

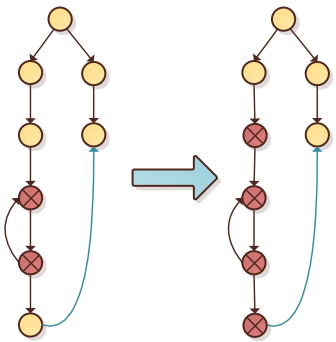

Figura 4.6.12. Ilustración del proceso de prohibición de nodos por niveles.

La Figura 4.6.12 muestra los nodos que detecta el algoritmo detección de ciclos y el algoritmo de prohibición por niveles.

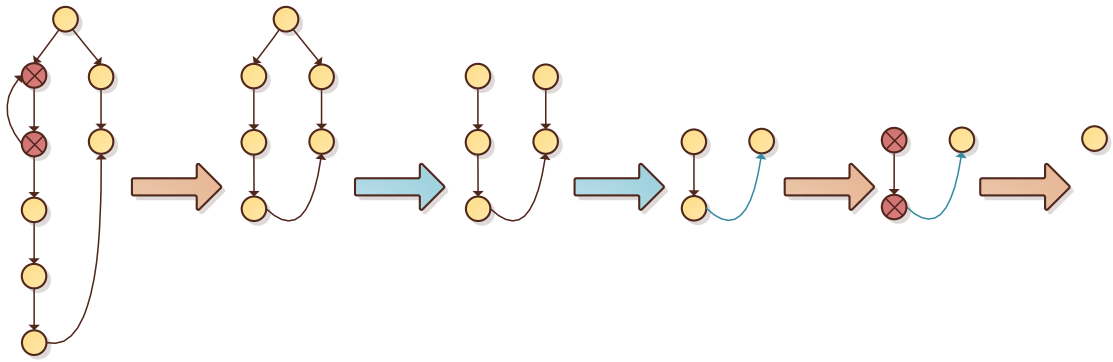

Figura 4.6.13. Ilustración del proceso de prohibición por niveles. Análisis de un caso concreto.

Los grafos presentan una estructura de árbol por niveles. En cada paso se poda un nivel del grafo. Antes de podar un nivel del grafo, se comprueba que el siguiente nivel no contiene posiciones prohibidas. Si existe una posición prohibida, se elimina del grafo y se suben a ese nivel sus hijos, volviendo a iterar hasta que no queden posiciones prohibidas en este nivel. Cuando se analiza el nivel a podar, se buscan los nodos que tienen algún padre. Cada uno de estos nodos determinará un camino de posiciones prohibidas cuyo origen será un nodo situado en el mismo nivel que se está analizando. Estas posiciones quedan prohibidas y se eliminan directamente del grafo. Las flechas en rojo de la Figura 4.6.13 muestra el caso de aplanamiento del nivel por eliminación de nodos prohibidos. Las flechas azules muestra la poda del subgrafo por niveles. La Figura 4.6.14 detalla la implementación del algoritmo.

```
lista<Element> prohibicionPorNiveles(estado: State, prohibidos: lista<Element>) {
   grafo := copia(grafoDe(estado))
   listaNivel := posicionesNucleares(estado)

   Mientras listaNivel no sea vacía {
      listaNivelSiguiente := listaVacía()
      Para cada elemento de listaNivel {
         listaHijos := hijosDe(grafo, elemento)
         Para cada elemento-hijo de listaHijos {
            Si posiciónProhibida(elemento-hijo) {
               listaHijosTmp := hijosDe(grafo, elemento-hijo)
               listaHijos := unión(listaHijos, listaHijosTmp)
               Eliminar elemento-hijo de grafo
               Eliminar elemento-hijo de listaHijos
            }
         }
         listaNivelSiguiente := unión(listaHijos, listaNivelSiguiente)
         Eliminar elemento de grafo
      }

      Para cada elemento de listaNivelSiguiente {
         listaPadres := padresDe(grafo, elemento)
         Para cada elemento-padre de listaPadres {
            Añadir elemento-padre a prohibidos
            Si listaNivelSiguiente contiene a elemento-padre {
               Eliminar elemento-padre de listaNivelSiguiente
            }
            listaPadresTmp := padresDe(grafo, elemento-padre)
            listaPadres := unión(listaPadres, listaPadresTmp)
         }
      }
      listaNivel := listaNivelSiguiente
   }
   Devolver prohibidos
}
Nota: La listaNivel nunca contendrá posiciones prohibidas.
```

Figura 4.6.14. Algoritmo de prohibición de nodos por niveles.

Obsérvese que el algoritmo de detección de ciclos puede fusionarse con el algoritmo de prohibición por niveles, para obtener una única etapa. Sin embargo, se ha preferido no realizar dicha fusión hasta que no se disponga de una versión completa del algoritmo global, momento en el cuál se podrán estudiar todas las etapas en conjunto, a fin de sintetizar un algoritmo más óptimo.

4.6.4 Asignación de marcadores

En esta sección se detalla los distintos algoritmos que intervienen en el proceso de cálculo de marcadores, destacando el concepto de particiones de nodos. Una partición representa un conjunto de elementos o nodos pertenecientes a un estado del autómata LALR(1) con posiciones no prohibidas. Estos elementos se relacionan entre sí debido a que representan posiciones donde debe aparecer un mismo marcador.

Debe indicarse que, debido a que el proceso de detección de posiciones prohibidas es, actualmente, heurístico, también será heurístico el proceso de asignación de marcadores. La consecuencia práctica es que, para algunas gramáticas patológicas, el resultado de la

asignación de marcadores dé lugar a gramáticas que no son LALR(1) (la Figura 4.6.15 muestra un ejemplo de gramática que no puede ser marcada adecuadamente por nuestros algoritmos). Afortundamente, los casos patológicos son raros. Actualmente el equipo de investigadores del proyecto Santander/UCM PR34/07-15865 tienen acotado el problema y trabajan en la obtención de una solución que funcione para cualquier gramática LALR(1) (incluso para gramáticas patológicas, como la mostrada en la Figura 4.6.15).

S ::= A # B
A ::= X b
A ::= Y a
B ::= X a
B ::= Y b
X ::= D
Y ::= E
D ::= c
D ::= λ
E ::= c d
E ::= λ

Figura 4.6.15. Una gramática LALR(1) patológica. El marcado de esta gramática mediante los procesos descritos en esta memoria produce una gramática que no es LALR(1).

4.6.4.1 Partición de Nodos en los estados

Para definir estas particiones se hace uso de un procedimiento descendente por niveles dirigido por la función *primerosCjtoPre(elemento, posición).* Los pasos que realiza el algoritmo de particiones se basa en un procesamiento de conjuntos de nodos o elementos clasificados por niveles. Considerando un grafo de un estado del AF LALR, se calcula:

- Caso básico: El nivel inicial del que parte el algoritmo se compone de las posiciones nucleares, nodos que no se han obtenido mediante un cierre-lambda en el grafo considerado del estado. El conjunto de elementos que compone un nivel no puede incluir posiciones prohibidas, restricción que cumplen de manera inmediata aquellas posiciones determinadas como nucleares.

- Caso no básico: Los restantes niveles se calculan como conjunto de nodos asociados a posiciones no nucleares, nodos que se han obtenido mediante un cierre-lambda en el grafo considerado, y que se consideran por niveles definidos por una relación directa padre-hijo con los nodos del nivel anterior. Sin embargo, esta relación directa no se cumple cuando algún hijo corresponde a una posición prohibida. En este caso, se consideran los hijos de este nodo como elementos del nivel actual y se vuelve a considerar la situación.

Este proceso termina cuando que se llega a una situación en la que todos los nodos que se consideran para procesar en un nivel corresponden a posiciones no prohibidas. En este

momento se definen las particiones considerando para cada hijo del nivel a procesar el conjunto primerosCjtoPre(…), de forma que:

- Los nodos con conjuntos primerosCjtoPre(…) que no sean disjuntos, posean algún elemento en común, pertenecerán a una misma partición.
- Los hijos con conjuntos primerosCjtoPre(…) disjuntos con los conjuntos primerosCjtoPre(…) del resto de hijos pertenecerán a particiones independientes.

```
lista<lista<Element>> partition(estado: State) {
   grafo := grafoDe(estado)
   listaHijos := posicionesNucleares(grafo)
   listaParticiones := listaVacía() //lista que contiene listas de elementos
   listaSímbolosParticiones := listaVacía() //lista que contiene los símbolos
   asociados a una partición de listaParticiones, por orden de posiciones en
   ambas listas
   k := 0
   Mientras listaHijos no sea vacia {
      listaHijosTmp := listaVacía()
      Para cada elemento de listaHijos {
         listaSímbolos := primerosCjtoPre(elemento,posicionMarcada(elemento))
         fusionar := falso
         Desde k hasta tamañoLista(listaParticiones) {
            listaSímbolosTmp := listaSímbolosParticiones[k]
            Si ¬disjunto(listaSímbolos, listaSímbolosTmp) {
               listaSímbolos := unión(listaSímbolos, listaSímbolosTmp)
               Añadir elemento a listaParticiones[k]
               fusión = cierto
            }
         }
         Si ¬fusión {
            AñadirÚltimo listaSímbolos a listaSímbolosParticiones
            AñadirÚltimo {elemento] a listaParticiones
         }
         //Calcular siguiente nivel mediante aplanamiento: no considerar hijos
         de posiciones prohibidas para el nivel pero si los hijos de estos
         como nivel actual.
         listaHijosTmp := hijosDe(grafo, elemento)
         listaHijosNueva := listaVacía()
         Para cada elemento-hijo de listaHijosTmp {
            Si posiciónProhibida(elemento-hijo) {
               listaHijosTmp2 := hijosDe(grafo, elemento-hijo)
               Para cada elemento-hijo2 de listaHijosTmp2 {
                  Si listaHijosTmp no contiene a elemento-hijo2 {
                     Añadir elemento-hijo2 a listaHijosTmp
                  }
               }
            } si no {
               Si listaHijosNueva no contiene a elemento-hijo {
                  Añadir elemento-hijo a listaHijosNueva
               }
            }
         }
      }
      k := tamañoLista(listaParticiones)
      listaHijos := listaHijosNueva
   }
   Devolver listaParticiones
}
```

Figura 4.6.16. Algoritmo de cálculo de particiones de un estado.

Este proceso genera en cada grafo de cada estado del AF LALR(1) un conjunto de particiones de nodos del grafo.

La Figura 4.6.16 detalla la implementación del algoritmo de particiones. En él ha de tenerse en cuenta las siguientes consideraciones:

- La variable listaParticiones se inicia vacía y, su cálculo final contendrá el resultado esperado del algoritmo.
- En la lista de símbolos de particiones listaSimbolosParticiones, cada posición de la lista determina el conjunto de símbolos de partición asociado a la partición de la lista de particiones ubicado en la misma posición.
- Los elementos de listaParticiones y listaSimbolosParticiones se relacionan por su posición en las listas.

4.6.4.2 Fusión de particiones

Una vez aplicado el cálculo de particiones a cada uno de los estados del autómata LALR(1) se obtiene un conjunto de particiones a nivel global. Este nuevo conjunto puede contener particiones cuyos elementos aparecen repetidos, ya sea total o parcialmente, en otra partición distinta. En este caso, se realiza una fusión de particiones que consiste simplemente en la unión de dichas particiones implicadas en una única partición. La Figura 4.6.17 muestra el algoritmo que realiza el proceso de fusión de particiones de manera eficiente.

```
lista<lista<Element>> globalPartition(estados: lista<State>) {
   listaParticiones := listaVacia()
   Para cada estado de estados {
      listaParticiones := union(listaParticiones, partition(estado))
   }

   listaFusionada := listaVacia()
   i := 0
   Repetir {
      añadirA(listaFusionada,obtenerPosicion(listaParticiones, 0))
      eliminarPosicion(listaParticiones, 0)
      listaElementos := obtenerPosicion(listaFusionada, i)
      Para cada elemento de listaElementos {
         k := 0
         Mientras k <= tamaño(listaParticiones) {
            listaElementosTmp := obtenerPosicion(listaParticiones, k)
            Si eliminarElemento(listaElementosTmp, elemento) {
               listaElementos := union(listaElementos, listaElementosTmp)
               eliminarPosicion(listaParticiones, k)
               k := k-1
            }
            k := k+1
         }
      }
      i := i+1
   } mientras ¬vacia(listaParticiones)

   Devolver listaFusionada
}

Nota: las listas son punteros.
```

Figura 4.6.17. Algoritmo de fusión de particiones.

Nótese que los símbolos asociados a las particiones y los símbolos de preanálisis de los elementos no se utilizan a la hora de realizar la unión, ya que este proceso involucra únicamente posiciones.

4.6.4.3 Marcado de posiciones para terminales

El proceso anteriormente descrito no tiene en cuenta la siguiente situación: si en un nivel aparece una posición que precede a un terminal, a fin de evitar un conflicto de desplazamiento-reducción, dicha posición deberá marcarse sucesivamente con los marcadores para las posiciones de los niveles inferiores asociados a elementos que incluyan el terminal entre sus símbolos de preanálisis. Esta situación se descubrió con posterioridad en el proceso de investigación, por lo que se ha añadido como una etapa adicional. Estos múltiples marcadores aparecerán únicamente precediendo a un terminal, por lo que serán almacenados en una estructura diferente.

```
{tabla, lista} marcadoAdicional(estados: lista<State>) {
   listaFusionada := globalPartition(estados)
   marcadoresAdicionales := tablaVacía()
   listaNumerosFusión := listaVacía()

Para cada estado de estados {
   listaParticiones := obtenerParticion(estado)
   i := 0
   Mientras i < listaParticiones.longitud {
      Para cada elemento de la forma [A → α.a β,Ω] de listaParticiones[i] {
         j := i+1
         nivel := 0
         Mientras j < listaParticiones.longitud {
            simbolosPartición := obtenerSímbolosParticion(listaParticiones[i])
            Si simbolosParticion contiene el símbolo a {
               numeroMarcador := posición(elemento, listaParticiones)
               Si nivel = listaMarcadoresAdicionales.longitud {
                  AñadirÚltimo marcadorDe(elemento, listaFusionada) a
                   marcadoresAdicionales[elemento]
               } Si no si listaMarcadoresAdicionales[nivel] ≠ numeroMarcador {
                   listaNumerosFusión := añadeOUne(núneroMarcador
                    listaMarcadoresAdicionales[nivel], listaNumerosFusión)
               }
               nivel := nivel+1
            }
            j := j+1
         }
      }
      i := i+1
   }
}
   Devolver {marcadoresAdicionales, listaNumerosFusión}
}

Nota: Las sublistas de listaNúmerosFusión son disjuntos entre sí.
Nota: las funciónes marcadorDe() y añadeOUne() corresponden a la Figura 4.6.19
```

Figura 4.6.18. Algoritmo de cálculo de marcadores adicionales para posiciones terminales.

La Figura 4.6.18 muestra el algoritmo que asigna marcadores adicionales a terminales. El algoritmo explota el hecho de que, en las listas de particiones calculadas en el proceso de

partición de los elementos en cada estado, las particiones aparecen ordenadas por niveles (primero las de los niveles inferiores, luego las de los niveles superiores).

```
----- Funciones auxiliares -----
posición marcadorDe(elemento: Element, listaFusionada) {
   i := 0
   Mientras i < listaFusionada.longitud {
      partición := listaFusionada[i]
      Si partición contiene a elemento {
         Devolver i
      }
   }
}

lista añadeOUne(número1, número2, lista ) {
   lista1 := listaVacía()
   lista2 := listaVacía()
   Para cada listai de lista {
      Si listai contiene a número1 {
         lista1 := listai
      }
      Si listai contiene a número2 {
         lista2 := listai
      }
   }
   Si lista1 y lista2 son vacías {
      Añadir {número1, número2} a lista
   } si no si lista1 ≠ lista2 {
      Si lista1 es vacía {
         Añadir número1 a lista2
      } si no si lista2 es vacía {
         Añadir número2 a lista1
      } si no {
        lista1 := unión(lista1,lista2)
        Eliminar lista2 de lista
      }
   }
   Devolver lista
}
```

Figura 4.6.19. Función auxiliar del algoritmo de cálculo de marcadores adicionales para posiciones terminales.

Cada particion de la *listaFusionada* nos indica un número de marcador correspondiente según su posición en dicha lista. El algoritmo calcula la tabla *marcadoresAdicionales* cuya clave es un elemento y cuyo valor es un conjunto de números de marcador correspondientes a la posición de las particiones de la *listaFusionada*, lo que determinan en orden, los marcadores que preceden al terminal de la posición marcada. La lista *listaNumerosFusión*, contiene listas que alojan números de marcador equivalentes entre sí. Esta información se utiliza a la hora de insertar adecuadamente los marcadores en el modelo XLOP.

Nótese por último que, como en el caso de los algoritmos de detección de ciclos y de prohibición por niveles, el marcado de posiciones para terminales podría fusionarse con los procesos de partición. No obstante, como en el caso previo, en este proyecto de Sistemas Informáticos se ha preferido mantener la organización en etapas para facilitar la optimización global del proceso, una vez que éste se haya caracterizado completamente.

4.6.4.4 Marcado de la gramática

Una vez finalizados todos los pasos anteriores, es necesario actualizar el modelo de objetos XLOP con la inserción de los nuevos marcadores. Utilizando la información de qué elementos han sido marcados con qué marcador o marcadores concretos, por cada elemento se accede a su atributo especial que guarda un puntero a la regla de producción original de la que fue derivada. De esta manera, se actualiza el modelo de objetos XLOP introduciendo marcadores en las reglas de producción y creando nuevas reglas de producción vacías correspondientes a los marcadores insertados.

En este proceso debe resolverse, no obstante, un inconveniente adicional. Por cada elemento se puede obtener la posición exacta donde se introduciría un marcador en la regla de producción original. Esta posición coincide con la posición del "punto" del elemento. Sin embargo, una vez que se introduce un marcador en la regla original, los terminales o no terminales del cuerpo de la regla son desplazados de su posición original. Ahora un nuevo elemento ya no podrá indicar la posición exacta donde se introduciría un nuevo marcador en la misma regla de producción original.

Este inconveniente se ha solucionado ordenando el conjunto de elementos marcados antes de ser recorridos, y, mientras se recorren, calculando el error de desplazamiento para realizar la siguiente inserción de forma correcta. Sin embargo, la ordenación de los elementos no es trivial. Los marcadores son representados mediante un número no asignado a priori. El conjunto de elementos marcados original no contiene elementos, sino conjunto de elementos. Cada conjunto de elementos indica un único número de marcador, que no es necesario asignar directamente. Debido a esto, las reglas de producción de las que han derivado los elementos de un mismo conjunto, no tienen por qué iguales. Por otra parte, se deberá tener en cuenta aquellos números de marcadores para los elementos que llevan asociados más de un marcador, cuya información está contenida en otra estructura diferente. En el cuerpo de la regla de producción original, los marcadores se insertan como instancias de la clase Marker. Esta clase posee un atributo ·unique que indica si dicho marcador aparece una y sólo una vez en toda la gramática para futuras referencias.

Para ordenar los elementos, poder conocer si son únicos en la gramática y asignarles un número concreto de marcador, se usa una estructura compleja pero eficiente en procesamiento. La estructura consiste en una tabla, cuya clave es una regla RuleClass y como valor, un árbol de búsqueda equilibrado. De esta manera, todos los elementos Element que derivan de la misma regla RuleClass se insertan en el mismo árbol equilibrado, cuyo valor de ordenación consiste en la posición marcada del elemento. Además, como información adicional asociada a cada nodo del árbol, se almacena el número concreto de marcador (en este punto se asignan números concretos a los marcadores) que marca la posición del elemento y se indica si el marcador aparece una y sólo una vez en toda la gramática

Una vez insertados los marcadores Marker en las reglas de producción de la gramática original, sólo queda como último paso, insertar las reglas de producción vacías de los propios

marcadores. Basta tan sólo con conocer el número total de marcadores insertados para crear todas las reglas de marcadores vacías. Estas reglas son una instancia de la clase Rule cuyo tipo de regla es *Empty_Marker*.

4.7 El Distribuidor

El módulo distribuidor asigna atributos y ecuaciones a los marcadores para obtener un modelo XLOP optimizado de la gramática original (Figura 4.7.1).

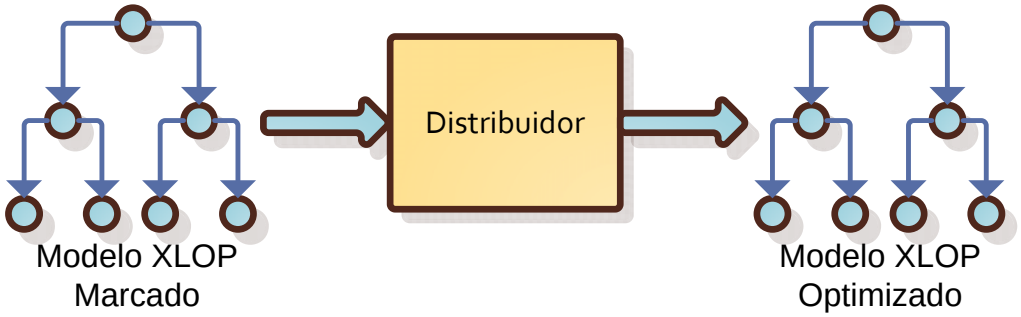

Figura 4.7.1. Módulo Distribuidor. Genera un modelo XLOP optimizado a partir de un modelo XLOP marcado.

El modelo XLOP optimizado posee atributos y ecuaciones semánticas asociadas a marcadores que permiten adelantar la ejecución de las operaciones y la disponibilidad de los datos. El efecto se consigue mediante una reordenación y simplificación de las expresiones semánticas asociadas a las reglas de producción de la gramática original. La optimización implica una disminución considerable en el tiempo de ejecución y consumo de memoria. El modelo XLOP optimizado sólo conserva aquellos marcadores utilizados en el alojamiento de atributos o ecuaciones que pueden optimizarse.

4.7.1 El proceso de optimización

El algoritmo de distribución de atributos y ecuaciones consiste en adelantar las operaciones que se ejecutan en unos momentos concretos de una gramática marcada a fin de optimizar la ejecución de la gramática. El proceso de optimización parte de un modelo XLOP marcado y cubre la generación de un modelo XLOP optimizado.

El proceso de optimización cubre dos objetivos:

- **Optimización espacial** / distribución de atributos: Consiste en evitar que la propagación de atributos heredados por reglas de copia en cadenas con recursión a izquierda cree estructuras en memoria que dependan de las longitudes de dichas cadenas. Implica una disminución en el consumo de memoria del sistema.

- **Optimización temporal** / distribución de ecuaciones: Consiste en evaluar las expresiones semánticas lo más pronto posible, sin necesidad de esperar a la reducción final de las producciones. Implica una disminución en el tiempo de ejecución.

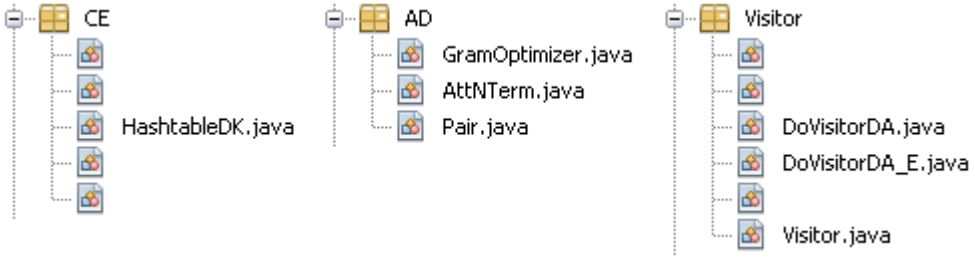

Figura 4.7.2. Desplegado de clases del módulo Distribuidor.

La Figura 4.7.2 presenta las diversas clases que intervienen en el proceso que realiza el módulo Distribuidor. Nótese que existe una clase Pair que facilita la gestión de dos valores relacionados entre sí.

4.7.1.1 Optimización espacial

La optimización espacial se basa en el análisis de las *ecuaciones de copia* que se aplican en cadenas generadas por las recursiones a izquierdas. Las ecuaciones de copia son ecuaciones de copia de valores de la forma X.a = Y.b, donde X e Y son no terminales, a y b sus respectivos atributos. Las ecuaciones de copia que interesan aparecen en posiciones prohibidas (véase sección 4.6.3).

Definimos la relación **>>** aplicada exclusivamente a atributos heredados de la siguiente manera:

- Sean X y B no terminales, a y b sus respectivos atributos heredados, *"X.a >> Y.b significa que existe una ecuación Y.b = X.a asociada con una producción X → Y α, siendo la posición X → .Y α prohibida".*

El cierre transitivo de esta relación, **>>**$^{+}$, indica la manera en que se propagan los valores de los atributos involucrados en las producciones que intervienen en cadenas generadas por recursión a izquierdas mediante ecuaciones de copia. Dicho cierre se define como:

- Sean X y B no terminales, a y b sus respectivos atributos heredados, *"X.a >>$^{+}$ Y.b significa que existe una cadena con recursión a izquierdas con un nodo X, otro Y, y en la que el valor del atributo a en X puede fluir hasta el valor del atributo b en Y propagado por ecuaciones de copia".*

De esta manera, se establece una relación entre atributos que se propagan mediante ecuaciones de copia.

4.7.1.1.1 Atributo optimizable

Se dice que un atributo heredado de un no terminal X.a es optimizable si:

- Cuando se define en una posición prohibida, lo hace a través de una ecuación de copia de la *forma* X.a = Y.b, con Y.b atributo heredado.
- Para todo *símbolo* Y que aparezca en posición no prohibida, y para cualquier par de atributos distintos c y d de Y, no puede ocurrir simultáneamente $Y.c >>^{+} X.a$ e $Y.d >>^{+} X.a$.
- Todo marcador de X marca una única ocurrencia de X en toda la gramática .
- Además, en dichas posiciones, para las ecuaciones utilizadas para calcular X.a, X.a = *e*, cualquier atributo sintetizada referido en *e queda a la izquierda de la posición*, y cualquier atributo heredado referido es, a su vez, optimizable .
- Si $Y.b >>^{+} X.a$, entonces Y.b debe ser también optimizable.

Utilizando la información que proporciona el cierre transitivo $>>^{+}$, es posible determinar qué atributos son optimizables Estos atributos se almacenarán en el marcador que precede al no terminal del comienzo de la cadena generada por la recursión a izquierdas, accediéndose a dicho marcador cuando sea necesario utilizar el valor de estos atributos. Así mismo, las ecuaciones de copia en posiciones prohibidas que involucren atributos optimizables podrán eliminarse, dado que ya no serán necesarias (los valores podrán tomarse directamente de los marcadores “que inician las cadenas con recursión a izquierda”).

4.7.1.2 Optimización temporal

La optimización temporal se basa en el análisis de las ecuaciones para el cálculo de los valores de los atributos heredados. Para ello se consideran los atributos sintetizados y heredados de los que depende el cálculo del valor de un atributo, y se localiza el primer marcador desde el cuál ya sean conocidos los valores de todos los atributos de los que dependen. El proceso de evaluación de la ecuación se lanza, entonces, en la producción asociada a dicho marcador. Aunque en el caso extremo podría ser que no fueran conocidos dichos valores hasta el momento de la reducción de la producción, en cuyo caso la ecuación no podrá evaluarse hasta dicho momento, en otros muchos casos el proceso de evaluación terminará mucho antes.

4.7.2 El paquete AD

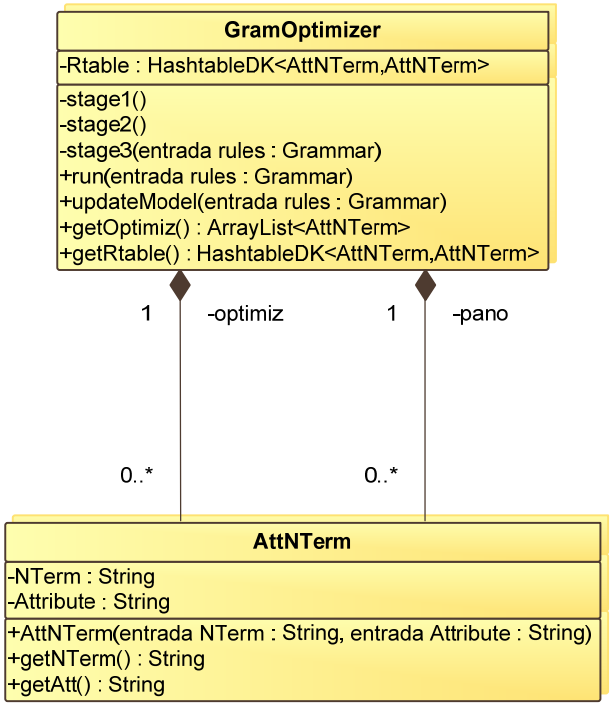

Figura 4.7.3. Diagrama de clases principales del proceso de optimización.

La Figura 4.7.3 presenta la clase más importante GramOptimizer, encargada de realiza las operaciones de optimización temporal y espacial sobre el modelo XLOP marcado. Cada uno de los métodos *stage1(), stage2(),* y *stage3()* implementan los algoritmos que se detallarán en la sección 4.7.3 de optimización espacial. El método *updateModel()*, además de calcular los atributos que pueden ser optimizados de manera que se beneficien de una optimización temporal, actualiza el modelo XLOP marcado produciendo el modelo XLOP optimizado, o lo que es lo mismo, la gramática optimizada esperada. El método *run()* realiza una llamada, en el orden correcto, a cada método mencionado.

4.7.3 Algoritmos

El algoritmo de optimización espacial se realiza en tres etapas secuenciales:

- Etapa I. Cálculo de la relación >>.
- Etapa II. Cálculo del cierre transitivo >>+.
- Etapa III. Cálculo de atributos optimizables.

4.7.3.1 Cálculo de las relaciones >>/>>+ y otra información auxiliar

4.7.3.1.1 Etapa I. Cálculo de la relación >>

```
Para cada posición prohibida X->.Yα {
   listaAtributosHeredados := atributos heredados de Y
   Para cada ecuación de la regla X->.Yα {
      Si el atributo se calcula como Y.h = X.h
         Eliminar el atributo h de listaAtributosHeredados
         Añadir Y.h a ·R>>[X.h]
   }
   Apilar atributos de listaAtributosHeredados en ·pano de la forma Y.h
}

Para cada no terminal Y marcado con M de la forma X -> αM.Yβ {
   Añadir Y a ·notForNTerms sin repeticiones
   Si Y ya ha sido marcado por M {
      Apilar atributos heredados de Y en ·pano de la forma Y.h
   } si no {
      Anotar Y marcado por M
      Para cada ecuación de la regla X -> αM.Yβ {
         Si existe un e tal que Y.h = e y e refiere algún atributo en Yβ {
            Apilar Y.h en ·pano
         }
      }
   }
}
```

Figura 4.7.4. Algoritmo de cálculo de la relación >>.

El algoritmo de la Figura 4.7.4 realiza el cálculo de la relación **>>** (sección 4.7.1.1). La implementación del algoritmo se realiza en el método privado *stage1()* de la clase GramOptimizer.

- Como entradas de esta etapa se requiere:
  - Tabla de símbolos de la gramática (véase 4.5.2): para la obtención de la lista de atributos heredados de un no terminal específico.
  - Lista de elementos que ocupan posiciones prohibidas (sección 4.6.3).
- Como resultado de esta etapa se obtiene:
  - ·notForNTerms: indica qué no terminales ocupan posiciones no prohibidas. Dicha información se almacena en el atributo privado ·notForNTerms de la clase GramOptimizer.
  - R>>: tabla que dado un atributo heredado X.a, determina el conjunto de atributos heredados Y.b tal que X.a >> Y.b (Y.b $\in$ R>>[X.a] si X.a >> Y.b). Inicialmente todas sus posiciones son vacías. Dicha información se almacena en el atributo privado ·Rtable de la clase GramOptimizer.

- ·pano: pila de atributos heredados que son no optimizables. Inicialmente será vacía. Dicha información se almacena en el atributo privado ·pano de la clase GramOptimizer con objetos AttNTerm**.**

El cálculo se enfoca en la obtención de las siguientes propiedades:

- Para cada atributo heredado se obtiene el conjunto de atributos heredados con los que se mantiene una relación **>>.**
- La relación entre marcadores y no terminales a los que marcan.
- El conjunto de atributos heredados que no pueden ser optimizables por incumplir algunas de las condiciones de ser atributo optimizable (véase sección 4.7.1.2).

Recuerde:

- Los conjuntos formados por las posiciones prohibidas y las posiciones marcadas siempre son disjuntos entre sí.

4.7.3.1.2 Etapa II. Cálculo del cierre transitivo >>$^+$

```
Para cada clave de ·R>>
   Para cada valor de ·R>>[clave]
      Si valor ≠ clave
         Si la entrada ·R>>[valor] no existe {
            Crear entrada ·R>>[valor]:{}
         } si no {
            Para cada valor-c ·R>>[valor] {
               Si ·R>>[clave] no contiene a valor-c {
                  AñadirÚltimo valor-c a ·R>>[clave]
         }
```

Figura 4.7.5. Algoritmo de cálculo cierre transitivo >>$^+$.

El algoritmo de la Figura 4.7.5 realiza el cálculo del cierre transitivo **>>**$^+$ (véase sección 4.7.1). La implementación del algoritmo se realiza en el método privado *stage2()* de la clase GramOptimizer. Nótese que el algoritmo actualiza destructivamente la tabla R>> con el cierre calculado.

4.7.3.2 Cálculo de atributos optimizables

El algoritmo de la Figura 4.7.6 determina qué atributos son optimizables. La implementación del algoritmo se realiza en el método privado *stage3()* de la clase GramOptimizer. En esta fase se recorren las estructuras de datos calculadas en las fases

anteriores para recuperar aquellos atributos que cumplen ser optimizables. La visita de las reglas (marcado mediante ***** en la Figura 4.7.6), para la detección de atributos heredados considerados optimizables que ya no cumplen dicha propiedad, se realiza mediante el recorrido del modelo de objetos XLOP con el patrón Visitor; concretamente, mediante la clase DoVisitorDA. La clase apila en ·pano los nuevos atributos heredados que dejan de ser optimizables.

```
Para cada no terminal Y de ·notForNTerms
   Para cada atributo heredado h de Y
      Para cada atributo heredado siguiente h' de Y
         Apilar en ·pano ·R>>[Y.h] ∩ ·R>>[Y.h']

Repetir {
   Visitar reglas(*) y apilar en ·pano atributos que dejan de ser optimizables
   Mientras ·pano no esté vacía
      Obtener cima
      Si ·optimiz contiene la cima {
         Eliminar cima de ·optimiz
         Apilar los valores de ·R>>[cima] en ·pano
      }
} Mientras ·pano no esté vacía
```

Figura 4.7.6. Algoritmo de cálculo de atributos optimizables.

- Como entradas de esta etapa se requieren las estructuras obtenidas de las etapas anteriores:
  - La pila ·pano de atributos no optimizables.
  - Información de qué no terminales ocupan posiciones no prohibidas de ·notForNTerms.
  - La tabla R>> con el cálculo del cierre transitivo **>>**$^+$ de los atributos.
- Como resultado de esta etapa se obtiene:
  - ·optimiz: conjunto de atributos optimizables. Inicialmente contiene todos los atributos heredados asociados a cada no terminal. Dicha información se almacena en el atributo privado ·optimiz de la clase GramOptimizer con objetos AttNTerm.

4.7.3.3 Algoritmo de optimización temporal

El algoritmo de optimización temporal se realiza en tres etapas secuenciales. Dicho proceso se realiza mediante el recorrido del modelo de objetos XLOP con el patrón Visitor; concretamente mediante la clase DoVisitorDAyE.

4.7.3.3.1 Etapa I. Redistribución de atributos y ecuaciones. Fase de cálculo

Esta primera fase recopila los datos sobre qué atributos y ecuaciones pertenecientes a las distintas reglas de producción de la gramática se pueden optimizar, y en qué marcador se puede alojar. Se realiza un recorrido del modelo de objetos XLOP de la gramática marcada y se asignan ecuaciones a marcadores a partir de la información sobre atributos optimizables obtenida en la fase anterior.

```
Si X.h es optimizable
   Si existe un marcador M inmediatamente a la izquierda de X
      Añadir par <ecuación actual, regla actual> a ·markAt[M]
   Si no apilar par <ecuación actual, regla actual> en ·removable

Si X.h es un atributo heredado {
   Buscar el primer marcador existente en β (Y->aXβ)
    que permita alojar la ecuación (*)

   Crear objeto con esta información: posición del marcador en
    la regla actual, regla actual y equación actual
   Guardar información en la pila: · stckMarkEq
}
```

Figura 4.7.7. Algoritmo de clasificación de expresiones semánticas como optimizables.

El algoritmo de la Figura 4.7.7 determina qué expresiones semánticas pertenecientes a cada regla de producción se pueden optimizar mediante su alojamiento en un marcador determinado. El algoritmo que detalla la figura pertenece a la operación que se realiza al recorrer una ecuación Equation del modelo XLOP marcado. Para la operación señalada con un asterisco **(*)** se realiza un cálculo considerando que: para poder alojar una ecuación en un marcador dentro de la misma regla, todos los atributos que intervengan en la ecuación deberán estar disponibles en el momento en que se ejecute dicho marcador.

En esta etapa se requiere la lista de atributos optimizables calculada mediante el algoritmo de optimización espacial. También se requiere la lista de todos los atributos heredados que pertenecen a cada no terminal, lista que se obtiene de la tabla de símbolos.

- Como resultado de esta fase se obtiene:

  - ·markAt: tabla que, dado un marcador, determina qué expresión semántica tiene asignado a dicho marcador y la regla de donde proviene. Por tanto ·markAt[M] es un conjunto de pares de la forma <ecuación, regla actual>.

  - ·removable: pila que determina las expresiones semánticas a eliminar de manera directa.

- ·stckMarkEq: pila que contiene objetos cuya información indica qué ecuaciones pueden ser optimizadas en qué marcador en concreto de la regla actual.

4.7.3.3.2 Etapa II. Redistribución de ecuaciones. Aplicación

En esta fase se realiza la reestructuración del modelo de objetos de la gramática XLOP marcada. Las expresiones semánticas contenidas en la tabla ·markAt de la Figura 4.7.7 desaparecen de sus producciones originales y se insertan como expresiones semánticas del marcador asignado. Las ecuaciones de copia en las que intervienen atributos heredados optimizables, dado que ya no son necesarias y el valor del atributo que copian ya se encuentra asignado a un marcador, se eliminan de sus producciones originales. Esta información la aporta los elementos de la pila ·removable. Para la información contenida en ·stckMarkEq perteneciente a ecuaciones semánticas, se comprueba que el marcador asignado es válido para alojar la ecuación. Si se incumple por ser un marcador no único en la gramática, se busca otro marcador de más a la derecha si existe. Por último, se eliminan los marcadores vacíos no utilizados de la gramática. Todo este proceso de modificación del modelo XLOP se realiza mediante el método updateModel*()* de la clase GramOptimizer.

4.8 El Generador

El generador de código produce un archivo escrito en el lenguaje de especificación CUP, cuya compilación produce una implementación específica del procesador en base a la gramática XLOP definida por el usuario. Por otro lado, el generador produce aquellas clases definidas en el lenguaje de programación Java que completan la especificación realizada en el archivo CUP.

La generación del código se realiza a partir de la información almacenada en el modelo de objetos de XLOP optimizado.

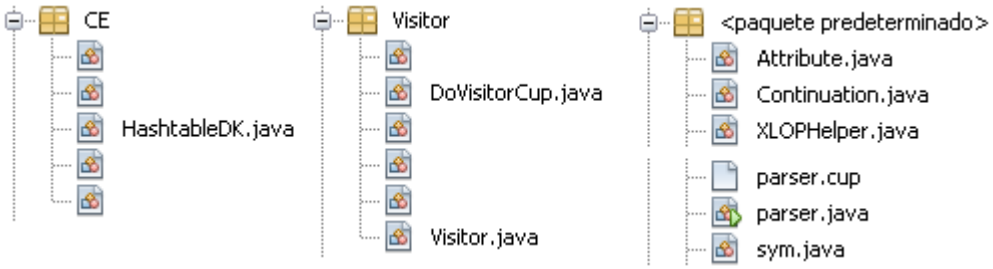

Figura 4.8.1. Desplegado de clases del módulo Generador.

La Figura 4.8.1 presenta las diversas clases que intervienen en el proceso que realiza el módulo Generador. El <paquete predeterminado> es el paquete que contiene todas las clases

que forman el Procesador (sólo se muestran las ilustrativas para esta sección). Las clases parser y sym son generadas por CUP a partir del fichero parser.cup.

4.8.1 Proceso de generación de código

Para la generación de código CUP se utiliza la información contenida en el modelo de objetos de XLOP correspondiente a la gramática optimizada. El código se genera mediante un recorrido del modelo de objetos con la ayuda del patrón Visitor.

La clase DoVisitorCup genera todos los ficheros necesarios que dependen de la información contenida en el modelo de objetos. Estos ficheros constituyen la principal componente del procesador resultante. El objetivo de la clase DoVisitorCup es generar el archivo parser.cup.

```
import java_cup.runtime.*;
import org.xml.sax.Attributes;

/* Preliminaries to set up and use the scanner. */

scan with {:
   Symbol sym = getScanner().next_token();
   return sym;
:}

parser code {:
   public static void main(String[] args) throws InterruptedException {
      ScannerS analizador = new ScannerS(args[0]);
      analizador.start(new ParserSAX(), Thread.MAX_PRIORITY);
      parser parserObj = new parser(analizador, new DefaultSymbolFactory());
      try {
         parserObj.parse();
      } catch (Exception e){
         e.printStackTrace();
         System.exit(-1);
      }
   }
:}

action code {:
   DesCifrador semObj = new DesCifrador();
:}
```

Figura 4.8.2. Contenido de archivo parser.cup. Sección de inicialización del proceso.

La Figura 4.8.2 presenta la estructura de una sección del archivo parser.cup, ejemplificada con la aplicación de cifrado introducida en el capítulo anterio. En dicha sección es importante realizar las siguientes operaciones:

- Bloque *scan with*: Se ejecutará cada vez que se lea un símbolo de entrada, especificándose aquí, por tanto, cómo y de dónde obtener dicho símbolo en

formato Symbol. Concretamente la operación se realiza mediante el método *getScanner().next_token()* cuyo tipo de scanner se especifica en el bloque *parser code*.

- Bloque *parser code*: Sección de declaración de los métodos y funciones que se ejecutarán en el arranque del parser. En este bloque se especifica y se configura el scanner a utilizar, concretamente con la implementación de parser SAX proporcionada por el entorno de ejecución de XLOP (véase sección 4.9).

- Bloque *action code*: Sección de declaración de variables globales del sistema. En este bloque se almacena en el atributo ·semObj una instancia de la Clase Semántica que contiene los métodos definidos por el usuario. Esto permite disparar cualquier método especificado por el usuario en un momento concreto en la ejecución de la gramática.

```
  /* Terminals (tokens returned by the scanner) */

terminal Attributes _O_Desc, _O_Dir, _O_I, _O_is;
terminal _C_Desc, _C_Dir, _C_is, _C_I;
terminal String Content;

  /* Non terminals */

non terminal SemanticDesc_Prima Desc_Prima;
non terminal SemanticDesc Desc;
non terminal SemanticSents Sents;
non terminal SemanticSent Sent;
non terminal SemanticInsts Insts;
non terminal SemanticInst Inst;
non terminal Semantic_M_1 _M_1;
non terminal Semantic_M_5 _M_5;
non terminal Semantic_M_7 _M_7;
non terminal Semantic_M_13 _M_13;
non terminal _M_17;

  /* The grammar */

Desc_Prima ::= Desc
{:
   RESULT = new SemanticDesc_Prima();
:};

Desc ::= _O_Desc: _O_Desc0 _M_1 Sents: Sents0 _C_Desc
{:
   RESULT = new SemanticDesc();
   final SemanticSents a3 = Sents0;
   final SemanticDesc a0 = RESULT;
```

Figura 4.8.3. Contenido del archivo parser.cup. Sección de declaración de variables.

La Figura 4.8.3 presenta la estructura del archivo parser.cup en cuanto a la sección de reglas de producción:

- Bloque de declaración de terminales: En este bloque se declaran los tipos de Tokens que serán devueltos por el scanner.

  - Mediante "*terminal Attributes*" se declaran los Tokens que pueden almacenar atributos. Este grupo está formado por los Tokens etiqueta de apertura exclusivamente.

  - Mediante "terminal" se declaran los Tokens restantes que no almacenan atributos. Este grupo está formado por los Token etiqueta de cierre.

  - *String* Content es el objeto que aloja el contenido de los textos XML o #pcdata leído en un momento concreto durante el proceso de un archivo XML de entrada.

- Bloque de declaración de no terminales: En este bloque se especifica mediante "*non terminal*" los registros semánticos asociados a los no terminales que alojan atributos heredados o sintetizados. Algunas de estos registros semánticos pueden tener un contenido complejo, especialmente los que refieren a marcadores. Sin embargo, la clase DoVisitorCUP se encarga de la generación automática de cada una de ellas.

- Bloque de gramática: Este bloque contiene las reglas de producción de la gramática XLOP traducido al lenguaje CUP. La complejidad del archivo reside en este bloque, que se detallará más adelante.

En los siguientes apartados se concreta el modelo a seguir en la generación del código CUP.

4.8.1.1 Código generado en función del tipo de objeto a evaluar

A continuación se describe la manera en la que se generan cada uno de los fragmentos de código de los que se compone el bloque de la gramática de un archivo CUP.

La definición sintáctica de las reglas de producción se escribe de manera casi idéntica al lenguaje de especificación de XLOP salvo un añadido: los símbolos que tienen al menos un atributo sintetizado o heredado aparecen acompañados de una variable descrita tras dos puntos, después de los mismos. Cada variable alojará un valor tras reducirse la regla de producción correspondiente, para el caso de los no termianales, o tras reconocerse el token,

```
Desc ::= _O_Desc: _O_Desc0 _M_1 Sents: Sents0 _C_Desc
{:
   RESULT = new SemanticDesc();
   final SemanticSents a3 = Sents0;
   final SemanticDesc a0 = RESULT;
   final Attributes a1 = _O_Desc0;
```

Figura 4.8.4. Regla de producción en CUP.

para el caso de los terminales.

Es necesario asignar los valores que devuelven los no terminales a variables finales *aX* para poder utilizarlos más adelante, como se muestra en la Figura 4.8.4. En esta figura puede verse cómo la etiqueta de apertura *_O_Desc* posee al menos un atributo. En concreto habíamos definido dos en nuestro ejemplo (*path* y *newpath*). Al ser una etiqueta de apertura cuyos atributos se leen del archivo XML de entrada, la gestión de dichos atributos se realiza mediante una instancia que implementa la interfaz SAX *Attributes* (ver [Brownell 2002]). Esta instancia se refiere desde la variable, asociada con la etiqueta de apertura *_O_Desc, _O_Desc0.*

```
class SemanticSents {
   Attribute<Object> file;
   Attribute<Object> fileh;
   public SemanticSents() {
      file = new Attribute<Object>();
      fileh = new Attribute<Object>();
   }
}
```

Figura 4.8.5. Registro semántico del no terminal Sents.

El no terminal *Sents* posee al menos un atributo. En concreto habíamos definido dos (*file* y *fileh*). Al ser un no terminal, la gestión de sus atributos se realiza mediante una instancia de la clase *SemanticSents,* como sugiere la Figura 4.8.5.

Por último, la cabeza de la producción debe generar una instancia de su registro semántico *SemanticDesc*. En todas las ecuaciones se asignará esta instancia a la variable local RESULT. RESULT es el valor que se obtiene al reducir una regla de producción. Es posible que necesitemos utilizar alguno de los atributos de la cabeza de la producción. Para ello, también es necesario asignar el valor RESULT a una variable final, en este caso, a0.

4.8.1.2 Los registros semánticos

Los registros semánticos asociados a los no terminales que aparecen en la gramática son generados de manera automática por la clase DoVisitorCup. Para cada no terminal, que aloje atributos sintetizados o heredados, es necesario crear un registro semántico asociado que permita gestionar sus atributos. Estas clases son nombradas con el prefijo "S*emantic"* seguido del nombre del no terminal que implementan. El objetivo de estas clases es permitir el alojamiento y el acceso a los valores contenidos en los atributos sintetizados o heredados que han sido definidos en la gramática XLOP.

Como se puede observar en la Figura 4.8.5**,** para el no terminal *Sents* sobre el que se han especificado dos atributos se genera un registro semántico nombrado *SemanticSents* donde se definen ambos atributos con carácter público de acceso y modificación. Los atributos soportan dos métodos:

- Para establecer el valor de un atributo se utiliza el método set(Object o) de la clase Attribute. El valor que se puede almacenar puede ser de cualquier tipo, al no existir todavía tipado.

- Para recuperar el valor de un atributo se utiliza el método get() de la clase Attribute. El valor que devuelve puede ser de cualquier tipo, al no existir todavía tipado estático en XLOP.

De esta manera, para establecer el valor del atributo old, utilizaríamos la instrucción old.set(objeto). Para recuperar el valor, objeto = old.get().

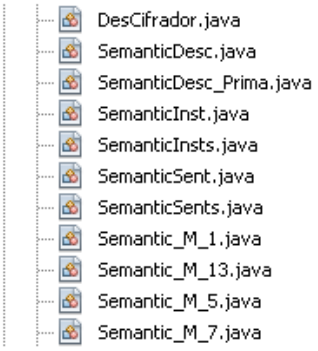

Figura 4.8.6. Clase Semántica Descifrador junto a los registros semánticos generados.

La Figura 4.8.6 presenta la Clase Semántica del ejemplo Descifrador junto a los registros semánticos generados por la clase DoVisitorCup. Estas clases se corresponden al <paquete predeterminado> de la Figura 4.8.1.

4.8.1.3 Traducción de las expresiones semánticas

Las expresiones semánticas se transforman en secuencias de operaciones según el tipo de expresión y la disponibilidad de los datos que presentan.

```
Inst ::= <I> #pcdata <is/> #pcdata </I> {
   file of Inst = addReplacementPair(text of #pcdata(0), text of #pcdata(1), fileh of Inst)
}
```

Figura 4.8.7. Ejemplo de regla de producción en XLOP.

La asignación de valores pertenecientes a variables de la definición sintáctica de la regla de producción a nuevas variables finales aX sólo es necesaria cuando los accesos se realizan dentro de un bloque *.whenAvaliable(new Continuation() {public void next()})* como muestra la Figura 4.8.8. Cuando se necesita acceder al valor de un atributo, o establecer su valor, es posible que dicho valor todavía esté por calcularse. Por ello, mediante el método *.whenAvaliable(new Continuation() {public void next()})* perteneciente a cada atributo, permite esperar la ejecución del código introducido en su cuerpo del método *public void next()*.

```
Inst ::= _O_I Content: Content0 _O_is _C_is Content: Content1 _C_I
{:
   RESULT = new SemanticInst();
   final String a2 = Content0;
   final String a4 = Content1;
   final SemanticInst a0 = RESULT;

   a0.fileh.whenAvaliable (
      new Continuation() {
         public void next() {
            a0.file.set(XLOPHelper.invoke(semObj, "addReplacementPair",
                                          new Object []{a2,a4,a0.fileh.get()}));
         }
      }
   );
:};
```

Figura 4.8.8. Traducción de la regla XLOP de la Figura 4.8.7 a código CUP.

De esta manera, la instrucción XLOP *file of Inst = addReplacementPair(…, …, file of Inst)* de la Figura 4.8.7 se traduce a "cuando el atributo *fileh* de la producción *Inst* esté disponible… ejecútese la asignación *a0.file.set("valor")"* donde a0 vuelve a ser el registro semántico del no terminal *Inst.*

En la Figura 4.8.8 se observa una variable adicional RESULT. Esta variable contiene una instancia del registro semántico de la cabeza de la producción actual. En este momento tiene lugar la asignación de valores a los atributos especificados en el registro semántico mediante las operaciones que se plantean, como ilustra la Figura 4.8.7. RESULT es el valor que se obtiene al reducir esta regla de producción.

4.8.1.3.1 Traducción de una ecuación semántica

Las funciones o métodos referidos desde una gramática en XLOP, cuya implementación aparece contenida en la Clase Semántica, se ejecutan mediante un método de la clase XLOPHelper. Una función o un método se traduce en código CUP como una instrucción que contiene la información completa de:

- Una instancia de la clase semántica.
- El nombre del método a invocar.
- Los objetos que el método necesita como argumentos de entrada.

Para buscar un método dentro de la clase semántica, y poder aplicar sus argumentos, se utiliza el método *XLOPHelper.invoke(instancia de la clase semántica, nombre método, lista de argumentos)* perteneciente a la clase XLOPHelper. La instancia de la clase semántica se almacena en la variable *semObj*, variable definida al inicio de la aplicación mediante el bloque *action code* (véase sección 4.8.1). En la Figura 4.8.7, la expression semántica

addReplacementPair(text of #pcdata(0), text of #pcdata(1), fileh of Inst) se traduce como *XLOPHelper.invoke(semObj, "addReplacementPair", new Object [](a2,a4,a0.fileh.get())}* de la Figura 4.8.8.

4.8.1.4 Código especial de marcadores

Existen tres tipos de reglas de producción de marcadores:

- Marcadores que optimizan atributos.
- Marcadores que optimizan ecuaciones.
- Marcadores que optimizan atributos y ecuaciones.

Analizaremos las dos primeras. La última es una simple combinación de las anteriores.

4.8.1.5 Marcadores que optimizan atributos

Para los marcadores que optimizan atributos es necesario crear registros semánticos asociados que permitan el alojamiento y el acceso a los valores contenidos en los atributos que optimizan. Estas clases son nombradas con el prefijo "*Semantic*" seguido del nombre del marcador que implementan.

```
class Semantic_M_1 {
   private Object semObj;
   private Stack stack;
   private int top;
   private Attribute<Object> filehOfSents;

   public Semantic_M_1(Object semObj, Stack stack, int top) {
      this.stack = stack;
      this.top = top;
      filehOfSents = null;
      this.semObj = semObj;
   }

   public Attribute<Object> filehOfSents() {
      if (filehOfSents == null) {
         filehOfSents = new Attribute<Object>();
         String pathOf_0_Desc0 = ((Attributes) XLOPHelper.getAttrs(stack,top,0)).getValue("path");

         filehOfSents.set(XLOPHelper.invoke(semObj,"loadFile",new Object [](pathOf_0_Desc0}));
      }
      return filehOfSents;
   }
}
```

Figura 4.8.9. Registro semántico del marcador _M_1.

A diferencia de los registros semánticos asociados a los no terminales de la gramática original, los atributos que optimizan no son visibles y accesibles de manera pública. Los atributos optimizados se guardan en una posición específica de la pila semántica del parser

CUP. Por ello, cada atributo optimizado se compone de un único *método específico*, concretamente nombrado como <prefijo>*Of*<sufijo>, con prefijo *nombre del atributo optimizado* y con sufijo *nombre del no terminal* (véase el método *filehOfSents*() de la Figura 4.8.9). Este método permite recuperar el atributo optimizado de la pila semántica permitiendo el acceso y modificación del valor del atributo mediante los métodos *.set()* y *.get()* (sección 4.8.1.2). Los *métodos específicos* se generan de manera diferente según el tipo de ecuación que optimizan.

La constructora del registro semántico requiere como parámetros: la Clase Semántica que implementa los métodos definidos en XLOP (*semObj,* véase *bloque action code* de la sección 4.8.1) y el puntero a la pila semántica del parser CUP junto a la posición que ocupa el atributo en la pila.

Cuando se realiza la optimización de uno o varios atributos heredados, el marcador sólo conserva aquella ecuación donde se asigna un valor inicial al primer atributo heredado, desde el cuál dicho valor se irá propagando por el árbol de derivación hacia niveles inferiores. El valor inicial puede provenir de un atributo de un no terminal, de un terminal o de un atributo optimizable (véase sección 4.7.1.1). Cuando se ejecuta un marcador, los elementos habrán sido evaluados previamente por el parser CUP, por lo que permanecerán en la pila semántica. El tratamiento de ambos casos se realiza de manera distinta con la ayuda de un método concreto de la clase XLOPHelper que permite obtener el elemento en la pila semántica:

- Cuando el valor proviene de un terminal, por ejemplo, una etiqueta de apertura, el valor se recupera mediante el método *XLOPHelper.getAttrs(pila, tope, desplazamiento en pila).getValue(nombre atributo del no terminal).*

- Cuando el valor proviene de un no terminal, es posible que dicho valor aún no se haya calculado. En estos casos, se vuelve a utilizar la estructura *.whenAvaliable()* a fin de realizar la asignación de valores cuando todos estén disponibles. Sin embargo, puesto que los no terminales se encuentran ahora en la pila semántica, es necesario utilizar el método *XLOPHelper.gettAttrs(…)* para recuperar el atributo deseado.

4.8.1.5.1 Análisis para un caso terminal

```
Desc ::= <Desc> Sents </Desc> {
   fileh of Sents = loadFile(path of <Desc>)
   end of Desc = saveFile(·"Exito"·, newpath of <Desc>, file of Sents)
}

Desc = <Desc> _M_1 Sents </Desc> {
   end of Desc = saveFile(·"Exito"·,newpath of _O_Desc,file of Sents)
}
```

**Figura 4.8.10. La primera regla presenta una producción sin optimizar.
La segunda regla presenta la misma producción ya optimizada.**

En la Figura 4.8.10, la instrucción *fileh of Sents = loadFile(path of <Desc>)* desaparece en la regla optimizada, por haber sido optimizado en el marcador M1. Dicha instrucción ahora se contempla en el marcador asociado M1.

```
_M_1 = {
   fileh of Sents = loadFile(path of _O_Desc)
}
```

Figura 4.8.11. Regla de marcador M1.

El método específico que permite obtener el valor del atributo optimizado queda reflejado en el método *filehOfSents()* del registro semántico del marcador M1 de la Figura 4.8.11. Para establecer el valor final del atributo *fileh*, obsérvese que se realiza de manera análoga a las instrucciones contenidas en el archivo parser.cup, salvo por la manera de acceder a los atributos.

4.8.1.5.2 Análisis para un caso no terminal

```
Sents ::= Sents Sent{
   fileh of Sents(1) = fileh of Sents(0)
   fileh of Sent = file of Sents(1)
   file of Sents = file of Sent
}

Sents = Sents _M_5 Sent {
   file of Sents = file of Sent
}
```

**Figura 4.8.12. La primera regla presenta una producción sin optimizar.
La segunda regla presenta la misma producción ya optimizada.**

En la Figura 4.8.12, se realiza una asignación cuyo valor depende de un no terminal, en concreto del no terminal *Sents*. Los atributos *fileh* (en realidad sólo uno, puesto que el otro *fileh* se detectó como alias del primero), se detectaron como optimizables y se optimizaron estableciendo los accesos a sus valores mediante el marcador M5. Esto queda reflejado en la regla de marcador M5 de la Figura 4.8.13.

```
_M_5 = {
   fileh of Sent = file of Sents(1)
}
```

Figura 4.8.13. Regla de marcador M5.

El método específico que permite obtener el valor del atributo optimizado *fileh* queda de la siguiente manera:

```
public Attribute<Object> filehOfSent() {
   if (filehOfSent == null) {
      filehOfSent = new Attribute<Object>();

      ((SemanticSents) XLOPHelper.getAttrs(stack,top,0)).file.whenAvaliable (
         new Continuation() {
            public void next() {
               filehOfSent.set(((SemanticSents) XLOPHelper.getAttrs(stack,top,0)).file.get());
            }
         }
      );
   }
   return filehOfSent;
}
```

Figura 4.8.14. Método de acceso y cálculo del atributo heredado optimizable *fileh* of Sent.

Para establecer el valor final del atributo, obsérvese en la Figura 4.8.14 que se realiza de manera análoga a las instrucciones contenidas en el archivo parser.cup, salvo por la manera de acceder a los atributos.

Por último sólo queda definir el fragmento del código CUP asociado a este tipo de marcador, el cual no es más que la llamada a la constructora de las clases semánticas que hemos definido (Figura 4.8.15).

```
_M_1 ::=
{:
   RESULT = new Semantic_M_1(semObj,CUP$parser$stack,CUP$parser$top);
:};

_M_5 ::=
{:
   RESULT = new Semantic_M_5(semObj,CUP$parser$stack,CUP$parser$top);
:};
```

Figura 4.8.15. Código CUP de los marcadores M1 y M5.

4.8.1.5.3 Casos especiales

Es posible que se den casos especiales como muestra la Figura 4.8.16:

```
_M_7 = {
   fileh of Insts = fileh of Sent.
}
```

```
public Attribute<Object> filehOfInsts() {
   if (filehOfInsts == null) {
      filehOfInsts = new Attribute<Object>();

      XLOPHelper.getAttr("filehOfSent",stack,top,1).whenAvaliable (
         new Continuation() {
            public void next() {
               filehOfInsts.set(XLOPHelper.getAttr("filehOfSent",stack,top,1).get());
            }
         }
      );
   }
   return filehOfInsts;
}
```

Figura 4.8.16. Método específico de obtención del atributo heredado optimizable fileh of Insts, alias del atributo optimizable principal fileh of Sent.

En un método específico, es posible que los atributos a los que se necesitan acceder sean atributos optimizados, debido a ser un alias de otro atributo. Para la obtención de estos atributos optimizados, hemos definido a su vez un método especial: mediante el método *XLOPHelper.getAttr(nombre del método especifico del atrobuto optimizable principal, pila, tope, desplazamiento)* podemos obtener dichos atributos (véase Figura 4.8.16).

4.8.1.6 Marcadores que optimizan ecuaciones

No se necesitan definir registros semánticos para los marcadores que optimizan únicamente ecuaciones. El código CUP asociado a este tipo de marcador se basa en una instrucción de asignación cuyos valores se obtienen mediante accesos a la pila semántica del parser CUP. Véase la siguiente Figura 4.8.17.

```
_M_17 = {
   fileh of Inst = file of Insts(1)
}
```

```
_M_17 ::=
{:
   ((SemanticInst) ((java_cup.runtime.Symbol) CUP$parser$stack.elementAt(CUP$parser$top-2)).value).fileh.set(
      ((SemanticInsts) ((java_cup.runtime.Symbol) CUP$parser$stack.elementAt(CUP$parser$top)).value).file.get()
   );
:};
```

Figura 4.8.17. Traducción del código XLOP del marcador M7 a CUP.

El marcador M17 realiza un adelanto en el cálculo de la ecuación, en este caso, un simple atributo sintetizado *file* of *Insts*, como presenta la Figura 4.8.18.

```
Insts ::= Inst Insts {
   fileh of Insts(1) = fileh of Insts(0)
   fileh of Inst = file of Insts(1)
   file of Insts(0) = file of Inst
}

Insts = Inst _M_13 Insts _M_17 {
   file of Insts = file of Inst
}
```

Figura 4.8.18. La primera regla presenta una producción sin optimizar. La segunda regla presenta la misma producción ya optimizada.

4.8.1.7 Marcadores que optimizan atributos y ecuaciones

Los marcadores que optimizan atributos y ecuaciones son una mezcla de los casos anteriores. Simplemente, las reglas de marcadores del código CUP contienen los dos tipos de instrucciones anteriores.

4.9 El entorno de ejecución

El entorno de ejecución XLOP es una biblioteca con un conjunto de componentes de uso común en cualquier aplicación generada con XLOP.

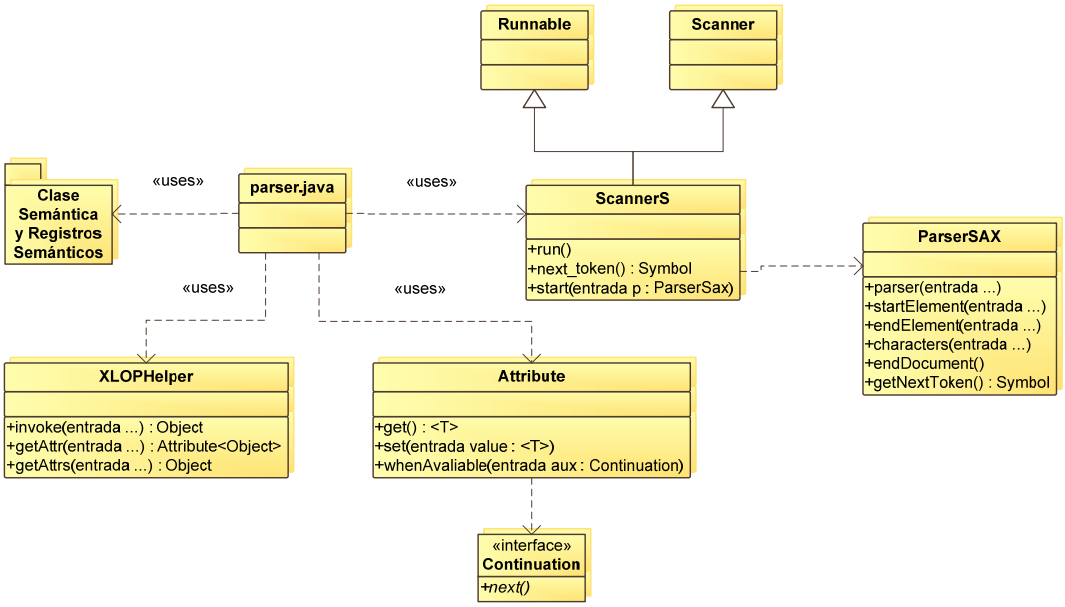

Figura 4.9.1. Relaciones entre clases del entorno de ejecución.

La Figura 4.9.1 esquematiza las clases contenidas en esta librería. A continuación se describe su estructura.

4.9.1 Conexión con un marco SAX

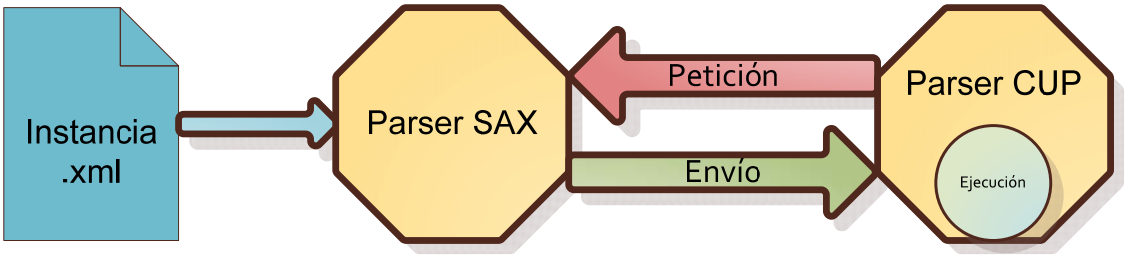

Figura 4.9.2. Proceso de intercambio de información de símbolos entre los dos hilos: Parser SAX y Parser CUP.

El análisis secuencial de un archivo XML se realiza a través de una implementación específica del parser SAX (véase sección 2.2.3.2). Por cada uno de los distintos tipos de elementos de los que se compone un fichero XML, SAX genera un evento específico. El parser CUP opera mediante la lectura de símbolos definidos por el compilador CUP. Mediante los eventos que genera el parser SAX, los elementos se traducen a los símbolos CUP junto a la información original que contienen los elementos XML. Existe, de esta forma, una incompatibilidad entre los regímenes de control de ambos componentes: tanto el parser SAX como el parser CUP son componentes que han de llevar el control. Es necesario, por tanto, invertir el control en uno de ellos. En Java, esto se consigue fácilmente usando hilos [Gosling et al. 2005].

De esta forma, el parser SAX y el parser CUP corren en hilos independientes, y se comunican a través de un buffer de tamaño 1. El parser CUP realiza la ejecución de los procedimientos hasta necesitar la lectura de un símbolo. En este momento, se realiza una petición de un símbolo al parser SAX. En cuanto el parser SAX realiza la lectura y traducción de un elemento XML a un símbolo, responde a la petición del parser CUP inmediatamente. Esto permite reanudar la ejecución del programa principal. Véase la Figura 4.9.2.

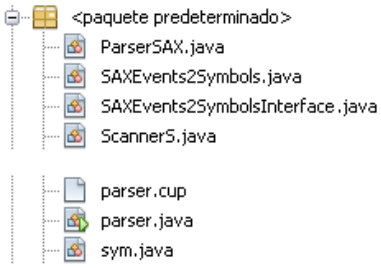

Figura 4.9.3. Desplegado de clases del entorno de ejecución.

La Figura 4.9.3 presenta las clases que intervienen en el entorno de ejecución. Con estas últimas clases, el <paquete predeterminado> queda completo, para el ejemplo descifrador, de la manera que muestra la Figura 4.9.4.

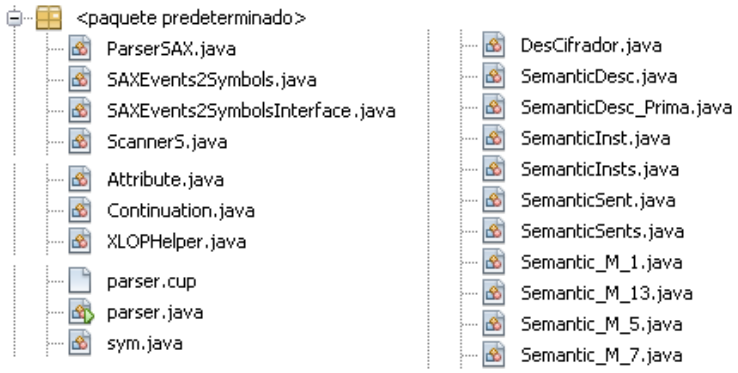

Figura 4.9.4. Contenido completo del <paquete predeterminado> de la aplicación Descifrador.

Ahora sólo es necesario incluir la lógica específica de la aplicación y las librerías que la aplicación requiera para completar enteramente la aplicación construida y generada mediante el entorno XLOP.

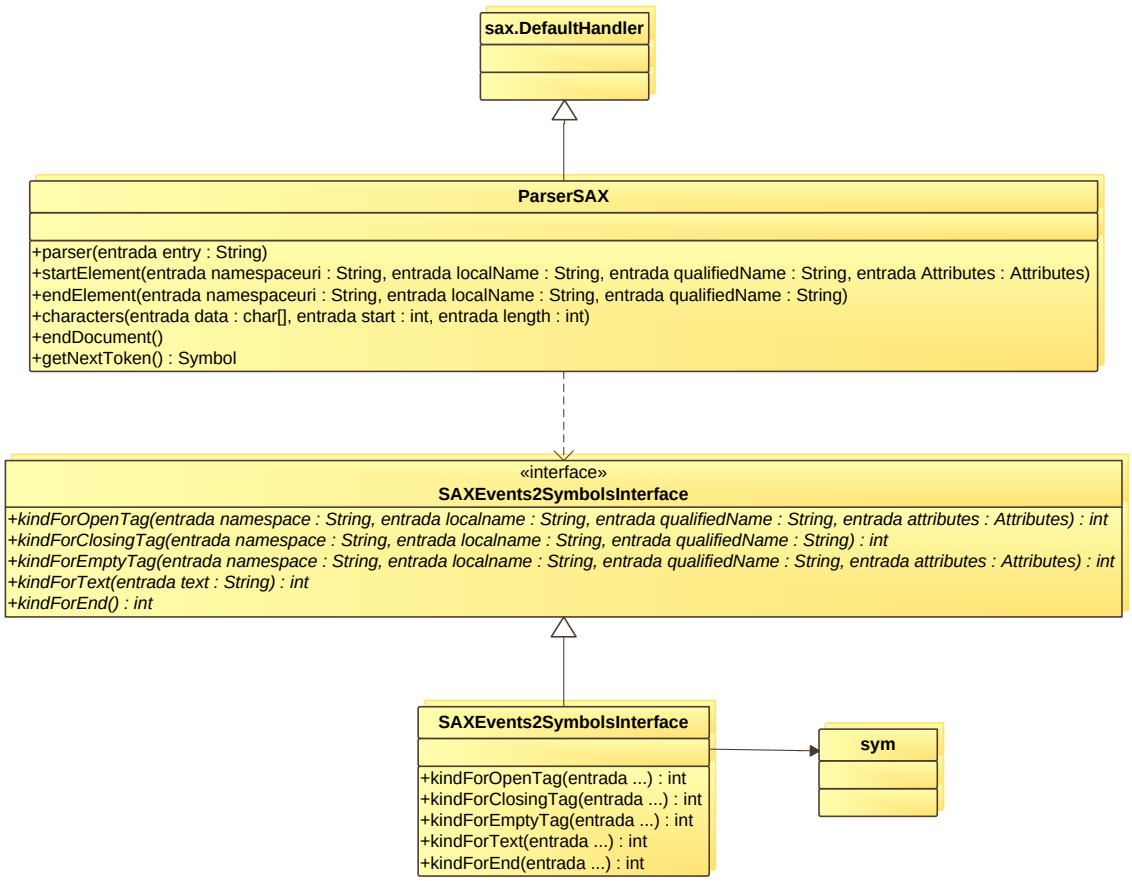

Figura 4.9.5. Interfaz para la traducción de elementos XML de SAX a símbolos CUP.

Por su parte, la traducción que realiza la implementación del parser SAX sobre los elementos XML se realiza mediante la implementación de los métodos definidos en la interfaz SAXEvents2SymbolsInterface como muestra la Figura 4.9.5.

La implementación de estos métodos de la interfaz SAXEvents2SymbolsInterface realiza la traducción de los elementos XML a símbolos CUP, como se puede observar en la Figura 4.9.6. La implementación de dicha interfaz se genera de manera automática gracias al método *SAXEvents2SymbolsText()* de la clase DoVisitorCup**.**

```
public class SAXEvents2Symbols implements SAXEvents2SymbolsInterface {

   @Override
   public int kindForOpenTag(String namespace, String localname, String qualifiedName, Attributes attributes) {
      if (qualifiedName.equals("Desc")) return sym._O_Desc;
      if (qualifiedName.equals("Dir")) return sym._O_Dir;
      if (qualifiedName.equals("I")) return sym._O_I;
      if (qualifiedName.equals("is")) return sym._O_is;
      return sym.error;
   }

   @Override
   public int kindForClosingTag(String namespace, String localname, String qualifiedName){
      if (qualifiedName.equals("Desc")) return sym._C_Desc;
      if (qualifiedName.equals("Dir")) return sym._C_Dir;
      if (qualifiedName.equals("is")) return sym._C_is;
      if (qualifiedName.equals("I")) return sym._C_I;
      return sym.error;
   }

   @Override
   public int kindForEmptyTag(String namespace, String localname, String qualifiedName, Attributes attributes) {
      throw new UnsupportedOperationException("Not supported yet.");
   }

   @Override
   public int kindForText(String text) {
      return sym.Content;
   }

   @Override
   public int kindForEnd() {
      return  sym.EOF;
   }
```

Figura 4.9.6. Traducción de eventos SAX en símbolos CUP.
Implementación de la interfaz SAXEvents2SymbolsInterface.

```
//----------------------------------------------------
// The following code was generated by CUP v0.11a beta 20060608
// Tue Apr 21 21:13:27 CEST 2009
//----------------------------------------------------

/** CUP generated class containing symbol constants. */
public class sym {
  /* terminals */
  public static final int _O_I = 4;
  public static final int _O_is = 5;
  public static final int Content = 10;
  public static final int _C_is = 8;
  public static final int _O_Desc = 2;
  public static final int EOF = 0;
  public static final int _C_I = 9;
  public static final int error = 1;
  public static final int _C_Desc = 6;
  public static final int _O_Dir = 3;
  public static final int _C_Dir = 7;
}
```

Figura 4.9.7. Tabla de traducción de los símbolos CUP a números enteros.

La definición de los símbolos CUP se realiza de manera automática por el compilador CUP mediante la generación de un archivo sym (véase Figura 4.9.7). Los símbolos CUP se identifican mediante números enteros.

Sin embargo, el parser CUP realiza peticiones de objetos de la clase Symbol. Un objeto de la clase Symbol, además de contener el número identificador, puede contener un objeto de cualquier tipo como valor. La clase ParserSAX realiza el parseo del documento XML recuperando etiquetas de apertura, de cierre, contenido y fin de archivo. Este parser es genérico para cualquier documento XML. El funcionamiento de la clase se puede ver en el método *Parser(fichero de entrada)*, en el cual se configura el parseo y se toma la fuente del documento. También hay que fijarse en los métodos *startElement(…), endElement(…), characters(…) y endDocument(…),* que son los que usa el parser SAX para saber qué debe hacer con cada etiqueta que va recuperando de la fuente. Un evento SAX, produce la ejecución de uno de los cuatro métodos anteriores, pero la traducción a un Symbol sólo se realiza cuando CUP realiza una petición. La traducción y contestación a dicha petición se realiza mediante el método *getNextToken*(). Sin embargo, existe una pequeña capa superior entre la clase ParserSAX y parser de CUP: el scanner o clase ScannerS. Esta capa permite lanzar una instancia de la clase ParserSAX como un hilo.

Por último, la conexión del scanner con el parser CUP se especifica en el archivo parser.cup, como ya hemos mencionado en la sección anterior.

4.9.2 Lógica de evaluación

Las operaciones que no son posibles de realizar en un determinado momento debido a la indisponibilidad de un dato se encolan a través de la clase Attribute y se lanzan en el mismo orden en que fueron encolados una vez que el dato ya se encuentra disponible. La clase Attribute es un contenedor de un dato, como bien sugiere la Figura 4.9.8.

El proceso se realiza de la siguiente manera:

- Las operaciones que se desean ejecutar en espera a que dato se encuentre disponible se codifican bajo la implementación del método *next()* de la interfaz Continuation. El objeto Continuation creado se encola a través del método *whenAvaliable(continuación)* de la clase Attribute.

- En el momento en que el dato ya esté disponible, momento que se conoce una vez que se realiza la asignación del valor del dato a través del método *set(valor)* de la clase Attribute, se ejecutan las operaciones en orden mediante la ejecución de cada método *next()* de cada objeto Continuation encolado.

- El método get() de la clase Attribute que proporciona el dato, se usa pues, cuando se conoce que el dato ya está disponible, es decir, se utiliza en la implementación del método *next()* de la interfaz Continuation.

En la sección 4.8.1.3 de Traducción de las expresiones semánticas se analizó el correcto funcionamiento que proporciona la codificación de operaciones bajo la implementación del método *next()* de la interfaz Continuation, y su encolamiento a través de instancias de la clase Attrbibute que representaban los atributos de los no terminales cuyo valor era desconocido en un momento de ejecución concreto de la aplicación.

```
/** Interfaz que implementa una operacion */
public interface Continuation {
   void next();
}

public class Attribute<T> {

   private T value;
   private ArrayList<Continuation> operations;
   private boolean available;

   public Attribute() {
      value = null;
      operations = new ArrayList<Continuation>();
      available = false;
   }

   public T get() {
      return value;
   }

   public void set(T value) {
      this.value = value;
      available = true;
      for (int i = 0; i < operations.size(); i++) {
         operations.get(i).next();
      }
   }

   public void whenAvaliable(Continuation aux) {
      if (available) aux.next();
      else this.operations.add(aux);
   }
}
```

Figura 4.9.8. Clase Attribute e Interfaz Continuation.

Capítulo 5
<e-Tutor>

5.1 Introducción a <e-Tutor>

En esta sección se ejemplifica el uso de XLOP en el desarrollo de una versión de <e-Tutor>, un sistema para el desarrollo de tutores socráticos basado en los trabajos de Alfred Bork y su equipo durante los ochenta [Bork 1985; Ibrahim 1989]. <e-Tutor> es un framework de creación de tutoriales interactivos. Permite presentar contenido de aprendizaje con preguntas interactivas y respuestas que dependen de las interacciones que el usuario va realizando conforme avanza en la aplicación.

Los tutoriales desarrollados en <e-Tutor> son *tutoriales socráticos*. Un tutorial socrático analiza las respuestas del estudiante a las preguntas planteadas, proporciona al mismo una realimentación apropiada, y decide el próximo paso a llevar a cabo en el proceso de aprendizaje. La realimentación dada se puede adaptar a distintos itinerarios de aprendizaje. En el caso más general el proceso de adaptación podría depender de la historia completa de la interacción del estudiante con el sistema, aunque en <e-Tutor> se adopta un mecanismo simple basado en contadores, al estilo de los trabajos de Bork [Bork 1985; Ibrahim 1989]. El sistema asocia contadores a cada posible respuesta de cada pregunta. Cada vez que el estudiante da una respuesta, incrementa el contador asociado. De esta forma, la realimentación y el siguiente paso a dar pueden depender del valor de dichos contadores. La Figura 5.1.1 muestra esquemáticamente un ejemplo de este tipo de tutorial. Los bocadillos representan puntos de respuesta donde el usuario debe proporcionar una respuesta a la pregunta realizada. Dichos puntos de respuesta están conectados con potenciales respuestas, que se encierran en una caja compartimentada. Las cajas moradas representan las realimentaciones. El flujo de aprendizaje se representa mediante flechas. Las etiquetas numéricas en las flechas que unen compartimentos de respuestas con realimentaciones indican los valores que deben tomar los contadores para que las flechas sean aplicables.

La característica principal de <e-Tutor> es que es altamente configurable. El contenido textual y la configuración de los diversos elementos que determinan la composición de los tutoriales se encuentra presente en los documentos XML. Debido a que cada tutorial se construye íntegramente mediante el contenido de un documento XML, el análisis y creación de los elementos que conforman los tutoriales pueden aprovechar las ventajas de especificación y optimización que el entorno XLOP aporta, obteniendo un procesador optimizado de documentos XML que permita construir cada uno de los tutoriales especificados en dichos documentos. De esta manera, en este capítulo se aborda la construcción de la aplicación <e-Tutor> mediante el entorno de desarrollo XLOP.

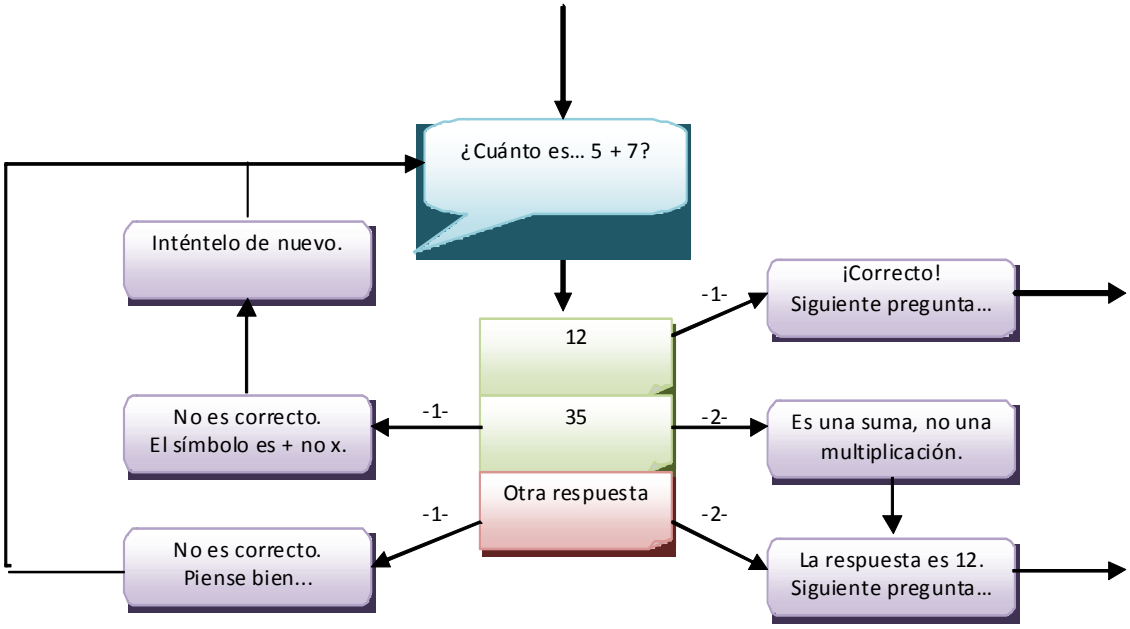

Figura 5.1.1. Fragmento de un tutorial socrático.

<e-Tutor> se desarrolló utilizando un enfoque precursor de XLOP para la programación del generador [Sierra et al. 2008b], y se ha utilizado como campo de pruebas para distintos experimentos relativos al desarrollo de sistemas e-Learning dirigidos por lenguajes [Sierra et al. 2007; Sierra et al. 2008a]. La actual refactorización mejora la mantenibilidad de la versión previa. Así mismo, se han añadido nuevas características y mejoras al sistema.

En este capítulo se describe la refactorización XLOP de <e-Tutor>. Por motivos de simplicidad, se omiten algunos elementos que muestran características similares a otras ya ilustradas. En el apéndice A pueden encontrarse las versiones completas de los componentes involucrados en el desarrollo XLOP de <e-Tutor>.

5.2 La lógica específica de la aplicación: El Marco de Aplicación en <e-Tutor>

<e-Tutor> se concibe como un sistema que, procesando un documento XML que describe un tutorial socrático, *instancia* automáticamente un marco de aplicación que caracteriza este tipo de tutoriales. De esta forma, desde el punto de vista de XLOP, este marco constituye la *lógica específica de la aplicación* en <e-Tutor>.

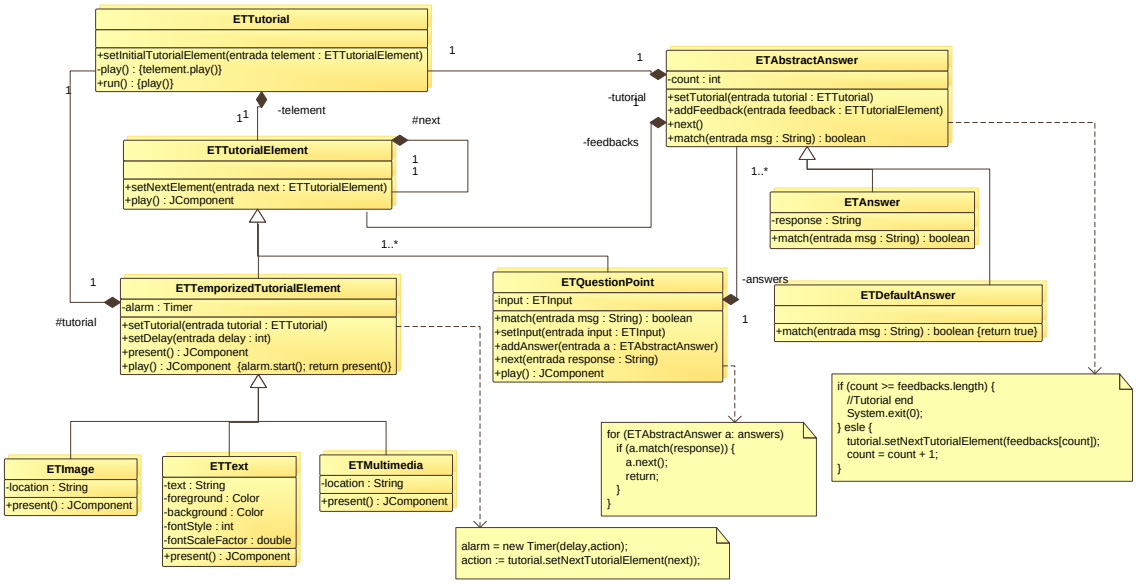

Figura 5.2.1. Marco de aplicación para representar tutoriales en <e-Tutor>.

La Figura 5.2.1 presenta las principales clases y métodos de los diversos componentes incluidos en el marco de aplicación <e-Tutor> para representar tutoriales. Todo tutorial empieza con una instancia de la clase ETTutorial, el objeto tutorial en sí, al cuál se añaden los distintos componentes. Esta clase principal, además de poseer el método que inicia la ejecución del tutorial, configura la manera en la que se visualizarán los componentes en la ventana de la aplicación. El tutorial comienza con la ejecución de un elemento inicial de tutorial, que se establece mediante el método *setInitialTutorialElement(elemento)*, y se ejecuta con el método *play()*. La ejecución de los siguientes elementos se realiza a través del vínculo que los une con el elemento anterior. Este vínculo es el atributo ·next, atributo que poseen por herencia, todos los elementos que derivan de la clase abstracta ETTutorialElement. La ejecución secuencial de los elementos que derivan de esta clase abstracta se realiza de manera temporizada, de tal forma que, una vez acabado el tiempo de espera, se ejecuta el siguiente elemento de la cadena. La clase abstracta ETTutorialTemporizedElement implementa dicha función a través del método *setDelay(tiempo)* y *play()*, como sugiere la Figura 5.2.1. Cada elemento concreto, deberá implementar el método *present()* que devuelva la implementación del componente para su inserción en la ventana de la aplicación. El método *play()* pues, es una llamada que el objeto tutorial realiza sobre cada elemento ETTutorialElement que posee en un momento concreto, el cuál se cambia por el siguiente elemento a través del disparo del temporizador (véase los comentarios de código de la Figura 5.2.1). Como elementos básicos del marco de aplicación de <e-Tutor> nos referiremos a las clases:

- ETText: Componente que muestra un mensaje de texto de estilo configurable.

- ETImage: Componente que muestra una imagen.

- ETMultimedia: Componente que muestra y reproduce contenido multimedia o vídeos. Este componente mostrado es una simplificación ideal de los diversos componentes multimedia que en realidad presenta la aplicación <e-Tutor>.

Las preguntas son también elementos ETTutorialElement que se disparan de manera temporizada, cuya estructura es la siguiente. La clase ETQuestionPoint posee una lista de respuestas, que pueden ser respuestas concretas ETAnswer o una respuesta por defecto ETDefaultAnswer (ambas subclases de ETAbstractAnswer). La clase pregunta opera de manera que, tras obtenerse una respuesta (configurando la entrada ·input) por parte del usuario, recorre en orden la lista de respuestas ETAbstractAnswer con el objetivo de encontrar una respuesta coincidente. Esta comprobación se realiza con el método *match(respuesta)* que implementan ETAnswer y ETAbstractAnswer. La respuesta coincidente, provocará la ejecución de su método *next()*. En este momento entran en juego los contadores y los feedbacks mencionados en la introducción del capítulo. Cada ejecución del método *next()* de una respuesta, incrementa su contador. Puesto que cada respuesta contiene una lista de feedbacks o nodos raíces de cadenas de elementos de tipo ETTutorialElement, cada ejecución del método *next()* de una respuesta, realizará la ejecución de un feedback diferente. Si no existen más feedbacks, o una cadena de elementos ETTutorialElement es discontinua, el tutorial llegará a su fin.

5.3 Los documentos XML y la DTD: El Lenguaje <e-Tutor>

<e-Tutor> incluye un lenguaje de marcado basado en XML para describir tutoriales como documentos XML, que, debidamente procesados, permiten generar y ejecutar dichos tutoriales.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Tutorial SYSTEM "tutorial.dtd">
<Tutorial start="p1" height="400" width="500" title="eTutor"
          logoPath="imgs/logo.jpg" color="FFFFFF" color2="F0EEEB">
      <Features>
            <Feature name="notAnswerErrorMsg">
                  There's not a matching answer
            </Feature>
            <Feature name="notAnswerErrorTitle">ERROR</Feature>
            <Feature name="notFeedbackErrorMsg">
                  There's not feedback
            </Feature>
            <Feature name="notFeedbackErrorTitle">ERROR</Feature>
      </Features>
      <Problem id="p1">
            <Text fontStyle="0" fontScaleFactor="1.5" colorf="ffffff"
                                                      colorb="123456">
                  ¿Cuánto es... 5 + 7?
            </Text>
            <QuestionPoint>
                  <Answer>
                        <Response>12</Response>
                        <Feedback next="p2">
                              <Image location="imgs/happy.jpg"/>
                              <Text delay="2000">
                                    ¡Correcto!Siguiente pregunta...
                              </Text>
                        </Feedback>
                  </Answer>
                  <Answer>
                      <Response>35</Response>
                        <Feedback next="p1">
                              <Text>
                                    No es correcto. El símbolo es + no x.
                              </Text>
                              <Text delay="2000">Inténtelo de nuevo.</Text>
                        </Feedback>
                        <Feedback next="p2">
                              <Image location="imgs/sad.jpg"/>
                              <Text>
                                    Es una suma, no una multiplicación.
                              </Text>
                              <Text delay="2000">
                                  La respuesta es 12. Siguiente pregunta...
                              </Text>
                        </Feedback>
                  </Answer>
                  <AnotherAnswer>
                        <Feedback next="p1">
                              <Text>No es correcto. Piense bien...</Text>
                        </Feedback>
                        <Feedback next="p2">
                              <Text>
                                 La respuesta es 12. Siguiente pregunta...
                              </Text>
                         </Feedback>
                  </AnotherAnswer>
            </QuestionPoint>
      </Problem>
      <Problem id="p2">
            <Text>Continuemos con...</Text>
      </Problem>
</Tutorial>
```

Figura 5.3.1. Codificación XML del fragmento del tutorial de la Figura 5.1.1.

La Figura 5.3.1 muestra la codificación XML del tutorial de la Figura 5.1.1 en este lenguaje. La estructura que siguen estos documentos XML se componen de una cabecera de configuración de aspectos ajenos al contenido del tutorial (p.ej., el texto informativo en caso de producirse un error durante la ejecución de la aplicación). Esta información aparece contenida entre las etiquetas <Features>. Los atributos que presentan las etiquetas permiten configurar los diferentes componentes, identificados a su vez, por el nombre significativo de la etiqueta. De esta manera, las dimensiones de la ventana del tutorial y los colores de fondo se configuran mediante los atributos *height*, *width*, *color1* y *color2.* El cuerpo de estos documentos XML contiene una secuencia de problemas contenidos entre las etiquetas <Problem>. Cada problema se identifica mediante su atributo *id*. Los problemas se estructuran como una secuencia encadenada de *elementos básicos <e-Tutor>*, que pueden finalizar con una pregunta <QuestionPoint>. Los elementos básicos <e-Tutor> son los textos <Text>, las imágenes <Image> y los contenidos multimedia <Multimedia>. Las preguntas <QuestionPoint> se componen de un conjunto de respuestas <Answer> y una respuesta por defecto <AnotherAnswer>. Las respuestas se disparan cuando el usuario introduce un valor que coincide con el campo <Response> de cada pregunta <Answer>. La secuencia de elementos que se presenta al disparar una pregunta corresponde al contenido de los <Feedback>. Cada <Feedback> posee un atributo *next* para indicar el siguiente problema a disparar cuando se termine la ejecución de todos sus elementos <e-Tutor>. Sin embargo, cada respuesta <Answer> contiene un conjunto de <Feedback> cuyo orden está asociado al contador que lleva cada respuesta, el cuál se incrementa en una unidad cada vez que el usuario responde de la misma manera, a la misma pregunta, permitiendo disparar un <Feedback> diferente en cada ocasión. Por último, la respuesta por defecto <AnotherAnswer> funciona de la misma manera que una respuesta <Answer>, salvo que no posee campo <Response> al dispararse siempre que una respuesta no coincida con las demás.

Todos los elementos básicos <e-Tutor> (salvo multimedia) contienen un temporizador para indicar el tiempo que la aplicación debe esperar hasta reproducir el siguiente componente. Este tiempo se da en milisegundos (ms) y se indica a través del atributo *delay* que poseen todas las etiquetas de dichos elementos. Por otra parte, dependiendo del tipo de componente, se puede configurar el formato del texto y los colores, o especificar las rutas de las imágenes y archivos multimedia a cargar. La Figura 5.3.2 muestra la DTD de los documentos XML, donde se pueden apreciar con mayor precisión, las diferentes opciones que se pueden aplicar a los distintos elementos de un tutorial.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % clrScr "cleanScreen (yes|no) #IMPLIED">
<!ENTITY % cBtn "continueBtn (yes|no) #IMPLIED">
<!ENTITY % sBar "controlBar (yes|no) #IMPLIED">
<!ENTITY % dg "dialog (yes|no) #IMPLIED
               dcolor CDATA #IMPLIED">
<!ENTITY % presentAttrs "title CDATA #REQUIRED
                         endBtn CDATA #IMPLIED
                         logoPath CDATA #IMPLIED
                         color CDATA #IMPLIED
                         color2 CDATA #IMPLIED
                         height CDATA #REQUIRED
                         width CDATA #REQUIRED">
<!ENTITY % textAttrs "colorf CDATA #IMPLIED
                      colorb CDATA #IMPLIED
                      fontStyle CDATA #IMPLIED
                      fontScaleFactor CDATA #IMPLIED">
<!ELEMENT Tutorial (Features?,Problem+)>
<!ATTLIST Tutorial start IDREF #REQUIRED %presentAttrs;>
<!ELEMENT Problem (QuestionPoint | ((Text|Image|Multimedia)+,QuestionPoint?))>
<!ATTLIST Problem id ID #REQUIRED>
<!ELEMENT Text (#PCDATA)>
<!ATTLIST Text delay NMTOKEN "1" %textAttrs; %clrScr;>
<!ELEMENT Image (#PCDATA)>
<!ATTLIST Image path CDATA #REQUIRED delay NMTOKEN "1" %clrScr;>
<!ELEMENT Multimedia (#PCDATA)>
<!ATTLIST Multimedia path CDATA #REQUIRED
                     height CDATA #REQUIRED
                     width CDATA #REQUIRED
                     %dg;
                     %cBtn;
                     %sBar;
                     %clrScr;>
<!ELEMENT QuestionPoint (Answer*,AnotherAnswer)>
<!ATTLIST QuestionPoint inputColumns NMTOKEN "10">
<!ELEMENT Feature (#PCDATA)>
<!ATTLIST Feature name CDATA #REQUIRED>
<!ELEMENT Answer (Response, Feedback+)>
<!ELEMENT AnotherAnswer (Feedback+)>
<!ELEMENT Response (#PCDATA)>
<!ELEMENT RespText (#PCDATA)>
<!ELEMENT Feedback ((Text|Image|Multimedia)+)>
<!ATTLIST Feedback next IDREF #IMPLIED>
<!ELEMENT Features (Feature)*>
```

Figura 5.3.2. DTD para las instancias XML <e-Tutor>.

5.4 La gramática XLOP: La Especificación XLOP del Generador de <e-Tutor>

Tal y como ya se ha indicado anteriormente, <e-Tutor> incluye un generador que, actuando sobre un documento XML que describe un tutorial, instancia el marco de aplicación <e-Tutor> para generar dicho tutorial, y, acto seguido, ejecuta el mismo. Este proceso puede describirse declarativamente en XLOP, tal y como se detalla en esta sección.

5.4.1 El estilo de la especificación

El generador de <e-Tutor> no es una aplicación trivial. De esta forma, tras ensayar una especificación puramente *funcional* del mismo, nos dimos cuenta de que la especificación resultante era excesivamente compleja, con demasiados atributos y ecuaciones. A fin de simplificar dicha especificación, hemos explotado un estilo de especificación avanzada en gramáticas de atributos. En este estilo, descrito en [Kastens 1991], las funciones semánticas se conciben como operaciones de un *tipo abstracto de datos* que mantiene explícitamente estado. Se usan, entonces, algunos atributos booleanos que representan cambios explícitos de estado, y que permiten mantener adecuadamente las dependencias entre computaciones.

El estado en sí se guarda como atributos de la clase semántica. Aunque esta técnica se puede emplear también con otro tipo de especificaciones XLOP, a fin de introducir optimizaciones, en el estilo seguido en <e-Tutor> la diferencia es que, cuando construimos la gramática, *somos conscientes de que la clase semántica representa estado*. En concreto, nuestra clase semántica introduce el siguiente estado:

- El tutorial (instancia de tipo ETTutorial).
- Una tabla de características, que asocia valores con las características especificadas mediante elementos de tipo Feature.
- Una tabla que asocia identificadores de problemas con los elementos que inician la secuencia de dichos problemas.
- Una tabla que asocia identificadores de problemas aún no creados con los elementos que deben *continuar* por los elementos iniciales de dichos problemas. Esta tabla se utilizará para fijar los elementos siguientes de los últimos elementos de los *feedbacks*. Esta tabla permite emplear una técnica similar a la conocida técnica de *parcheo* utilizada en la construcción de traductores en una única pasada [Aho et al. 2007].

En la especificación se usará una función sémática *after(e0,e1)* que devolverá siempre el valor de sus segundo argumento. Dado que la evaluación de expresiones semánticas en XLOP es puramente aplicativa, esta función será útil para introducir dependencias entre computaciones que cambian el estado de la clase semántica.

5.4.2 Procesamiento de las *características*

La Figura 5.4.1 muestra el fragmento de gramática XLOP asociado al procesamiento de las características (elementos *Features* y *Feature*). La función semántica *addFeature(…)* permite añadir una nueva característica. Nótese el significado de los atributo *featureStored* (la característica ha sido almacenada) y *featuresStored* (las características han sido almacenadas), como atributos que representan explícitamente la ocurrencia de cambios de estados en la

instancia de la clase semántica. El uso de la función *after(…)* asegura, de esta forma, que las características se añaden a la tabla en el orden en el que aparecen en el documento, y que, además, se añaden todas ellas una vez procesadas todas las características.

```
Features ::= Features Feature {
   featuresStored of Features(0) =
               after(featuresStored of Features(1), featureStored of Feature)
}

Features ::= Feature {
  featuresStored of Features = featureStored of Feature
}

Feature ::= <Feature> #pcdata </Feature> {
   featureStored of Feature = addFeature(name of <Feature>, text of #pcdata)
}
```

Figura 5.4.1. Reglas que modelan el procesamiento de las características.

5.4.3 Construcción del tutorial

```
Tutorial ::= <Tutorial> <Features> Features </Features> Problems </Tutorial> {
   tutorialCreatedh of Problems = after(featuresStored of Features,
      newTutorial(height of <Tutorial>, width of <Tutorial>,
      title of <Tutorial>, endBtn of <Tutorial>,
      newColor(color of <Tutorial>), newColor(color2 of <Tutorial>),
      logoPath of <Tutorial>))
   tutorialExecuted of Tutorial =
      ...
}
```

Figura 5.4.2. Regla de construcción del tutorial.

La Figura 5.4.2 muestra el fragmento de gramática XLOP que construye el tutorial. Nótese que el uso de *after(…)* asegura que la creación del tutorial depende de la tabla de características. La configuración de ventana y colores, entre otras opciones, se establecen mediante los atributos de la etiqueta <Tutorial>. El atributo heredado *turorialCreatedh* representa, precisamente, este hecho: el tutorial ha sido creado. El tutorial en sí estará referido por la instancia de la clase semántica.

5.4.4 Procesamiento de la lista de problemas

La Figura 5.4.3 muestra la caracterización XLOP del procesamiento de la lista de problemas. La función semántica *newProblem(…)* permite añadir un nuevo problema a la tabla de problemas. Esta función también realiza el *parcheo* de elementos cuyos elementos siguientes son el primer elemento del problema añadido. Nótese que, mediante el uso de *after(…)*, se garantiza que esto ocurrirá siempre después de que se haya creado el tutorial.

Obsérvese, también, la propagación del atributo *tutorialCreatedh* a través de toda la lista de problemas, y de cada problema individual. Por su parte, el atributo sintetizado *problemsCreated* representa el hecho de que los problemas han sido creados.

```
Problems ::= Problems Problem {
   tutorialCreatedh of Problems(1) = tutorialCreatedh of Problems(0)
   tutorialCreatedh of Problem = tutorialCreatedh of Problems(0)
   problemsCreated of Problems(0) = after(problemsCreated of Problems(1),
                                    newProblem(id of Problem, elem of Problem))
}

Problems ::= Problem {
   tutorialCreatedh of Problem = tutorialCreatedh of Problems
   problemsCreated of Problems = after(tutorialCreatedh of Problems,
                                    newProblem(id of Problem, elem of Problem))
}
```

Figura 5.4.3. Reglas de procesamiento de la lista de problemas.

5.4.5 Procesamiento de cada problema

Para procesar los problemas necesitamos, primeramente, modelar las reglas sintácticas para expresar sus modelo de contenidos (ver Figura 5.3.2). Efectivamente, en un problema:

- Puede aparecer un QuestionPoint sin más.
- O bien puede aparecer una secuencia (no vacía) de elementos normales, terminada, opcionalmente, con un QuestionPoint.

```
Problem ::= <Problem> ProblemDescription </Problem> {...}

ProblemDescription ::= QuestionPoint {...}

ProblemDescription ::= Element RestOfProblemDescription { ... }

RestOfProblemDescription ::= Element RestOfProblemDescription { ... }

RestOfProblemDescription ::= QuestionPoint { ... }

RestOfProblemDescription ::= { ... }
```

Figura 5.4.4. Estructura sintáctica para el tratamiento de elementos de un problema.

Las producciones que nos interesan para procesar esta estructura se muestran en la Figura 5.4.4. Nótese que utilizamos recursión a derechas (aunque ésta es tratada menos eficientemente por los analizadores ascendentes [Aho et al. 2007]) porque estamos interesados en distinguir explícitamente, en cada secuencia, el primer elemento de la secuencia. Efectivamente, el procesamiento que deseamos realizar es encadenar dichos elementos. Dicho procesamiento queda caracterizado en la Figura 5.4.5.

```
Problem ::= <Problem> ProblemDescription </Problem> {
    tutorialCreatedh of ProblemDescription = tutorialCreatedh of Problem
    id of Problem = id of <Problem>
    elem of Problem = elem of ProblemDescription
}

ProblemDescription ::= QuestionPoint {
    tutorialCreatedh of QuestionPoint = tutorialCreatedh of ProblemDescription
    elem of ProblemDescription = qp of QuestionPoint
}

ProblemDescription ::= Element RestOfProblemDescription {
    tutorialCreatedh of Element = tutorialCreatedh of ProblemDescription
    tutorialCreatedh of RestOfProblemDescription =
                              tutorialCreatedh of ProblemDescription
    elem of ProblemDescription =
                putNext(elem of Element, elem of RestOfProblemDescription)
}

RestOfProblemDescription ::= Element RestOfProblemDescription {
    tutorialCreatedh of Element =
                              tutorialCreatedh of RestOfProblemDescription(0)
    tutorialCreatedh of RestOfProblemDescription(1) =
                              tutorialCreatedh of RestOfProblemDescription(0)
    elem of RestOfProblemDescription(0) =
                putNext(elem of Element, elem of RestOfProblemDescription(1))
}

RestOfProblemDescription ::= QuestionPoint {
    tutorialCreatedh of QuestionPoint =
                              tutorialCreatedh of RestOfProblemDescription
    elem of RestOfProblemDescription = qp of QuestionPoint
}

RestOfProblemDescription ::= {
    elem of RestOfProblemDescription = voidElem()
}
```

Figura 5.4.5. Reglas de encadenamiento de elementos de un problema.

La función semántica *putNext(e1,e2)* realiza el encadenamiento (conceptualmente, dado dos elementos sin encadenar, produce un nuevo elemento con el primero encadenado al segundo). La función semántica *voidElem()* produce un elemento *nulo* (indicará el final de la ejecución del tutorial). El atributo *elem* sintetiza el elemento resultante.

5.4.6 Tratamiento de los elementos básicos

El tratamiento de los elementos básicos (texto, imagen, multimedia), es sencillo. La única precaución es hacerlo depender de la creación del tutorial. La Figura 5.4.6 muestra el fragmento de gramática XLOP que se encarga de ello. Por cada regla *Element* se crea un elemento <e-Tutor> con la configuración y contenido que presentan sus etiquetas de apertura, mediante los métodos concretos: *newText(…), newImage(…), newMultimedia(…)*. El elemento

creado se propaga sintetizado mediante el atributo *elem* de la cabeza de la producción para conexiones posteriores.

```
Element ::= <Text> #pcdata </Text> {
    elem of Element = after(tutorialCreatedh of Element,
          newText(text of #pcdata, delay of <Text>,newColor(colorf of <Text>),
                  newColor(colorb of <Text>), fontStyle of <Text>,
                  fontScaleFactor of <Text>, cleanScreen of <Text>))
}

Element ::= <Image/> {
    elem of Element = after(tutorialCreatedh of Element,
          newImage(path of <Image>, delay of <Image>, cleanScreen of <Image>))
}

Element ::= <Multimedia/> {
   elem of Element = after(tutorialCreatedh of Element,
          newMultimedia(path of <Multimedia>, height of <Multimedia>,
                width of <Multimedia>, dialog of <Multimedia>,
                newColor(dcolor of <Multimedia>), continueBtn of <Multimedia>,
                controlBar of <Multimedia>, cleanScreen of <Multimedia>))
}
```

Figura 5.4.6. Reglas de tratamiento de textos, imagenes y multimedia.

5.4.7 Tratamiento de los puntos de pregunta

```
QuestionPoint ::= <QuestionPoint> Answers AnotherAnswer </QuestionPoint> {
   tutorialCreatedh of Answers = tutorialCreatedh of QuestionPoint
   tutorialCreatedh of AnotherAnswer = tutorialCreatedh of QuestionPoint
   qph of Answers = after(tutorialCreatedh of QuestionPoint,
                          newQuestion(inputColumnsh of <QuestionPoint>))
   qp of QuestionPoint = addAnswer(qp of Answers, answ of AnotherAnswer)
}

Answers ::= Answers Answer {
   tutorialCreatedh of Answers(1) = tutorialCreatedh of Answers(0)
   tutorialCreatedh of Answer = tutorialCreatedh of Answers(0)
   qph of Answers(1) = qph of Answers(0)
   qp of Answers(0) = addAnswer(qp of Answers(1), answ of Answer)
}

Answers ::= Answer {
  tutorialCreatedh of Answer = tutorialCreatedh of Answers
  qp of Answers = addAnswer(qph of Answers, answ of Answer)
}
```

Figura 5.4.7. Reglas de tratamiento de puntos de pregunta.

La Figura 5.4.7 caracteriza el tratamiento de los puntos de pregunta (QuestionPoint). La función semántica *newQuestionPoint(…)* crea un nuevo punto de pregunta (nótese el uso de la función *after(…)*). El punto de pregunta se propaga hasta la primera respuesta utilizando el atributo heredado *qph*. A partir de aquí, se sintetiza el punto de pregunta que resulta de

añadir respuestas al mismo. La función semántica *addAnswer(…)* realiza dicho añadido (conceptualmente, dado un punto de pregunta y una respuesta, produce el nuevo punto de pregunta que resulta de añadir la respuesta al punto dado). El resultado se sintetiza en el atributo *qp*.

5.4.8 Tratamiento de las respuestas

```
Answer ::= <Answer> <Response> #pcdata </Response> Feedbacks </Answer> {
   tutorialCreatedh of Feedbacks = tutorialCreatedh of Answer
   answh of Feedbacks =
                after(tutorialCreatedh of Answer, newAnswer(text of #pcdata))
   answ of Answer = answ of Feedbacks
}

Feedbacks ::= Feedbacks Feedback {
  tutorialCreatedh of Feedbacks(1) = tutorialCreatedh of Feedbacks(0)
  tutorialCreatedh of Feedback = tutorialCreatedh of Feedbacks(0)
  answh of Feedbacks(1) = answh of Feedbacks(0)
  answ of Feedbacks(0) = addFeedback(answ of Feedbacks(1), elem of Feedback)
}

Feedbacks ::= Feedback {
   tutorialCreatedh of Feedback = tutorialCreatedh of Feedbacks
   answ of Feedbacks = addFeedback(answh of Feedbacks, elem of Feedback)
}
```

Figura 5.4.8. Reglas de tratamiento de una respuesta.

El tratamiento de cada respuesta sigue un patrón similar, tal y como se muestra en la Figura 5.4.8. La función *newAnswer(…)* crea una nueva respuesta, que se propaga mediante el atributo *answh* al primer elemento *Feedback*. A partir de aquí, mediante *addFeedback(…)* se van añadiendo las realimentaciones. El resultado se sintetiza en *answ*.

```
AnotherAnswer ::= <AnotherAnswer> Feedbacks </AnotherAnswer> {
   tutorialCreatedh of Feedbacks = tutorialCreatedh of AnotherAnswer
   answh of Feedbacks =
                after(tutorialCreatedh of AnotherAnswer, newDefaultAnswer())
   answ of AnotherAnswer = answ of Feedbacks
}
```

Figura 5.4.9. Regla de tratamiento de una respuesta por defecto.

Por su parte, el tratamiento de la respuesta por defecto es directo, como muestra la Figura 5.4.9.

5.4.9 Tratamiento de cada realimentación

```
Feedback ::= <Feedback>FeedbackElements<Feedback> { ... }

FeedbackElements ::= Element FeedbackElements { ... }

FeedbackElements ::= Element { ... }
```

Figura 5.4.10. Estructura de tratamiento de elementos de una realimentación.

El tratamiento de cada elemento *Feedback* depende, al igual que el tratamiento de cada elemento *Problem*, de una correcta caracterización de su modelo de contenidos. Esta vez, dicho modelo es una secuencia de elementos básicos. Por tanto, la estructura que interesa queda caracterizada por las reglas en la Figura 5.4.10.

```
Feedback ::= <Feedback> FeedbackElements </Feedback> {
  tutorialCreatedh of FeedbackElements = tutorialCreatedh of Feedback
  nexth of FeedbackElements = next of <Feedback>
  elem of Feedback = elem of FeedbackElements
}

FeedbackElements ::= Element FeedbackElements {
    tutorialCreatedh of Element = tutorialCreatedh of FeedbackElements(0)
    tutorialCreatedh of FeedbackElements(1) =
                        tutorialCreatedh of FeedbackElements(0)
    nexth of FeedbackElements(1) = nexth of FeedbackElements(0)
    elem of FeedbackElements(0) =
                        putNext(elem of Element, elem of FeedbackElements(1))
}

FeedbackElements ::= Element {
    tutorialCreatedh of Element = tutorialCreatedh of FeedbackElements
    elem of FeedbackElements = after(tutorialCreatedh of FeedbackElements,
                  linkWithProblem(elem of Element, nexth of FeedbackElements))
}
```

Figura 5.4.11. Reglas de tratamiento de elementos de una realimentación.

El procesamiento en sí se consigue de manera análoga al procesamiento de la lista de problemas (Figura 5.4.11). No obstante, es necesario propagar hasta el último problema el identificador del problema siguiente (valor del atributo *next* del elemento Feedback). Esto se consigue mediante un atributo heredado *nexth*. La función *linkWithProblem(…)* realiza el enlace de dicho elemento, o bien lo deja pendiente en la tabla de elementos pendientes para problemas, a fin de que pueda posteriormente enlazarse (en la llamada a *newProblem(…)* que cree el problema).

5.4.10 Activación del tutorial

```
Tutorial ::= <Tutorial> <Features> Features </Features> Problems </Tutorial> {
   ...
   tutorialExecuted of Tutorial =
      after(problemsCreated of Problems, run(getProblem(start of Tutorial)))
}
```

Figura 5.4.12. Regla de construcción del tutorial (ecuación semántica de activación).

La activación del tutorial se lleva a cabo, por último, tal y como muestra la Figura 5.4.12, una vez que todos los problemas han sido creados.

5.5 La Clase Semántica

La presente Clase Semántica se mostrará en varias figuras. Esta clase contiene la implementación, como métodos, de las diversas funciones semánticas contempladas en la sección anterior. Además de dichos métodos, el estado de la clase almacena la información de configuración, problemas, elementos pendientes (para resolver los conflictos mencionados) y el tutorial en sí, que se utilizan durante la creación de la instancia tutorial.

```
public class TutorialSemanticClass {

   private Hashtable<String,String> features;
   private Hashtable<String,ETTutorialElement> problems;
   private Hashtable<String,Stack<ETTutorialElement>> pendingElements;
   private ETTutorial tutorial;

   public TutorialSemanticClass() {
      features = new Hashtable<String,String>();
      problems = new Hashtable<String,ETTutorialElement>();
      pendingElements = new Hashtable<String,Stack<ETTutorialElement>>();
   }

   public boolean newTutorial(String height, String width, String title,
                              String endBtn, Color color, Color color2, String logoPath) {
      tutorial = new ETTutorial(Integer.valueOf(width).intValue(),Integer.valueOf(height).intValue());
      tutorial.setTitle(title);
      if (endBtn != null) tutorial.setLabelEndButton(endBtn);
      if (color != null) tutorial.setBlackboardColor(color);
      if (color2 != null) tutorial.setBlackboardColor2(color2);
      if (logoPath != null) tutorial.setLogoLocation(logoPath);
      return true;
   }

   public void run(ETTutorialElement e) {
      features = null; problems = null; pendingElements = null; System.gc();
      tutorial.setInitialTutorialElement(e);
      tutorial.run();
   }
```

Figura 5.5.1. Métodos de creación y arranque del tutorial.

La Figura 5.5.1 presenta el método principal de creación del objeto tutorial ETTutorial (*newTutorial(…)*), junto con el método (*run(…)*) de arranque del tutorial. A su vez, presenta las estructuras de datos previamente mencionadas. Más concretamente, ·features es una tabla que asocia mediante clave-valor, un tipo de error con su correspondiente mensaje. Dicha tabla se rellena mediante los valores que encierran las etiquetas <Feature> a través del método *addFeature(…)* (véase Figura 5.5.2). La tabla ·problems es una tabla que asocia mediante clave-valor, un identificador de problema con el primer elemento <e-Tutor> o nodo raíz del problema. Esta tabla se utiliza para almacenar los nodos raíces de los problemas y devolverlos cuando se realice una petición de un problema determinado. La tabla ·pendingElements asocia mediante clave-valor, un identificador de problema con una pila de elementos que deben conectarse con el nodo raíz del perteneciente al identificador de problema, que se encuentra disponible en la tabla ·problems. Cuando se cree realmente un problema con el método *newProblem(…)*, se utilizará la información de esta tabla ·pendingElements y ·problems para conectar correctamente los nodos almacenados en la pila, con el elemento raíz del problema correspondiente (véase Figura 5.5.5).

```
public boolean addFeature(String name, String text) {
   features.put(name,text);
   return true;
}

public Object after(Object a, Object b) {return b;}

public ETTutorialElement putNext(ETTutorialElement e, ETTutorialElement ne) {
   e.setNextElement(ne);
   return e;
}

public ETTutorialElement voidElem() {return null;}
```

Figura 5.5.2. Métodos de configuración y de enlace de elementos.

La Figura 5.5.2 presenta el método que permite conectar dos elementos entre sí, el método *putNext(elemento, elemento siguiente)*, así como el método *after(objeto1, objeto2)* que permite garantizar el secuenciamiento adecuado de los estados de la instancia de la Clase Semántica.

```
public ETTutorialElement newText(String text, String delay, Color foreground,
                                 Color background, String fontStyle, String fontScaleFactor,
                                 String cleanScreen) {
   ETText etext = new ETText();
   etext.setText(text);
   if (delay != null) etext.setDelay(Integer.valueOf(delay).intValue());
   else etext.setDelay(100);
   if (foreground != null) etext.setForegroundColor(foreground);
   if (background != null) etext.setBackgroundColor(background);
   if (fontStyle != null) etext.setFontStyle(Integer.valueOf(fontStyle).intValue());
   if (fontScaleFactor != null) etext.setFontScaleFactor(Double.valueOf(fontScaleFactor).doubleValue());
   if (cleanScreen != null) etext.cleanScreen(cleanScreen.equals("yes"));
   etext.setTutorial(tutorial);
   return etext;
}

public ETTutorialElement newImage(String path, String delay, String cleanScreen) {
   ETImage eimg = new ETImage();
   eimg.setLocation(path);
   eimg.setTutorial(tutorial);
   if (delay != null) eimg.setDelay(Integer.valueOf(delay).intValue());
   else eimg.setDelay(100);
   if (cleanScreen != null) eimg.cleanScreen(cleanScreen.equals("yes"));
   return eimg;
}

public ETTutorialElement newMultimedia(String path, String height, String width, String dialog,
                                       Color dcolor, String continueBtn, String controlBar, String cleanScreen) {
   ETMultimedia m = new ETMultimedia();
   m.setLocation(path);
   if (height != null) m.setHeight(Integer.valueOf(height).intValue());
   if (width != null) m.setWidth(Integer.valueOf(width).intValue());
   if (dialog != null) m.setDialog(dialog.equals("yes"));
   if (dcolor != null) m.setColor(dcolor);
   if (continueBtn != null) m.setContinueBtn(continueBtn.equals("yes"));
   if (controlBar != null) m.setControlBarVisible(controlBar.equals("yes"));
   if (cleanScreen != null) m.cleanScreen(cleanScreen.equals("yes"));
   m.setTutorial(tutorial);
   return m;
}
```

Figura 5.5.3. Métodos de creación de elementos <e-Tutor>.

La Figura 5.5.3 presenta los métodos concretos asociados a la creación de cada elemento básico de <e-Tutor>. Como puede observarse, estos métodos admiten valores no especificados o nulos, y devuelven el elemento básico que crean.

```
public ETQuestionPoint newQuestion(String inputColumns) {
   ETQuestionPoint qp = new ETQuestionPoint();
   if (inputColumns == null) inputColumns = "10";
   TextInput ti = new TextInput(Integer.valueOf(inputColumns).intValue());
   qp.setInput(ti);
   ti.setInputContinuation(qp);
   qp.setMsgNotAnswer(features.get("notAnswerErrorMsg"));
   qp.setMsgNotAnswerTitle(features.get("notAnswerErrorTitle"));
   return qp;
}

public ETAbstractAnswer newAnswer(Object answer) {
   ETAnswer a = new ETAnswer(answer);
   a.setTutorial(tutorial);
   a.setMsgNotFeedback(features.get("notFeedbackErrorMsg"));
   a.setMsgNotFeedbackTitle(features.get("notFeedbackErrorTitle"));
   return a;
}

public ETAbstractAnswer newDefaultAnswer() {
   ETDefaultAnswer a = new ETDefaultAnswer();
   a.setTutorial(tutorial);
   a.setMsgNotFeedback(features.get("notFeedbackErrorMsg"));
   a.setMsgNotFeedbackTitle(features.get("notFeedbackErrorTitle"));
   return a;
}

public ETQuestionPoint addAnswer(ETQuestionPoint qp, ETAbstractAnswer a) {
   qp.addAnswer(a);
   return qp;
}
public ETAbstractAnswer addFeedback(ETAbstractAnswer a, ETTutorialElement e) {
   a.addFeedback(e);
   return a;
}
```

Figura 5.5.4. Métodos de creación y vinculación de preguntas y respuestas.

La Figura 5.5.4 presenta los métodos de creación de una pregunta, una respuesta, y una respuesta por defecto. Más concretamente, *addAnswer(…)* permite añadir una respuesta ETAnswer o ETDefaultAnswer a una pregunta ETQuestionPoint. El método *addFeedback(…)* permite añadir un el elemento <e-Tutor> raíz de una cadena de elementos, a una respuesta ETAnswer o ETDefaultAnswer**.** Nótese que, a nivel de implementación, estos métodos actúan *destructivamente* (lo mismo que putNext, y que otros que serán introducidos más adelante). No obstante, debido al uso dado de los mismos en la especificación XLOP, este hecho no afecta el carácter declarativo de la especificación.

```
public boolean newProblem(String id, ETTutorialElement e) {
   Stack<ETTutorialElement> pendingElms = pendingElements.get(id);
   if (pendingElms != null)
      while(!pendingElms.empty())
         pendingElms.pop().setNextElement(e);
   problems.put(id,e);
   return true;
}

public ETTutorialElement getProblem(String id) {
   return problems.get(id);
}

public ETTutorialElement linkWithProblem(ETTutorialElement e, String problem) {
   if (problem == null) {  //Fin de tutorial.
      e.setNextElement(null);
      return e;
   }
   ETTutorialElement ne = problems.get(problem);
   if (ne != null) e.setNextElement(ne);
   else {
      Stack<ETTutorialElement> pendingElms = pendingElements.get(problem);
      if (pendingElms == null) {
         pendingElms = new Stack<ETTutorialElement>();
         pendingElements.put(problem,pendingElms);
      }
      pendingElms.push(e);
   }
   return e;
}
```

Figura 5.5.5. Métodos de creación de un problema y resolución de conflictos.

La Figura 5.5.5 presenta el funcionamiento que se sigue para resolver los conflictos que presenta el vincular un elemento perteneciente a un feedback con un problema todavía por crearse. Por último, hay que resaltar que la versión real de <e-Tutor> presenta muchos más elementos y métodos asociados que permiten desde la reproducción de vídeos de YouTube o contenidos Flash hasta la posibilidad de contestar preguntas pinchando sobre zonas de una imagen. Estas características adicionales se verán al final del capítulo, mediante una demostración de ejecución de un tutorial más completo y avanzado. En el apéndice A se encuentra el código completo de todos los componentes involucrados en el desarrollo.

5.6 Generación de la aplicación con XLOP

Para configurar el entorno XLOP y generar la aplicación final de <e-Tutor>, es necesario indicar la ruta de la Clase Semántica (sección 5.5) y la DTD (sección 5.3) en la lista de *archivos de usuario* de la pestaña de *Generación de archivos y procesamiento XLOP (*Figura 5.6.1*)*. La

lógica específica de los componentes que constituyen el marco de aplicación <e-Tutor> (sección 5.2), están disponibles como librería en el archivo eTutor.jar. Por otra parte, para la reproducción de contenidos multimedia es necesario añadir librerías adicionales también a la lista de *librerías de usuario*.

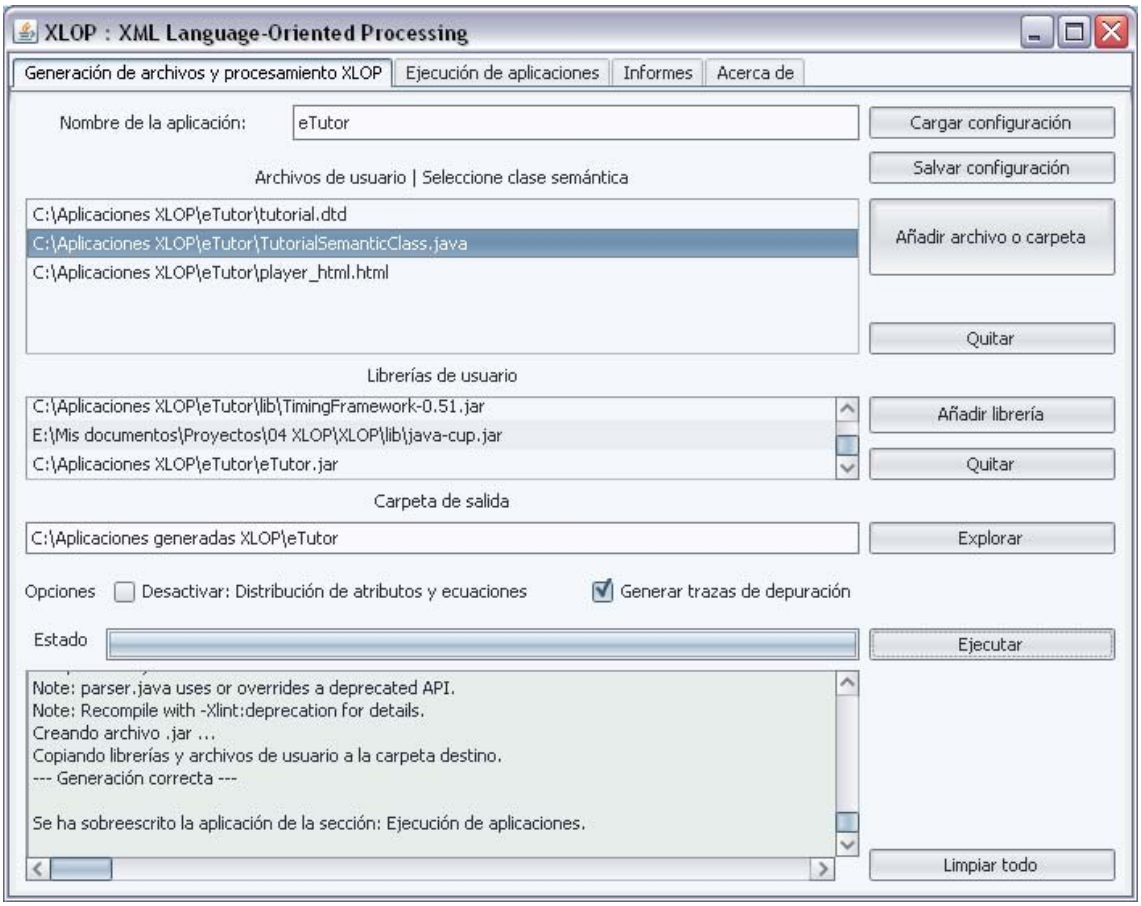

Figura 5.6.1. Procesamiento de la gramática <e-Tutor> y generación de la aplicación.

Mediante el botón *Ejecutar* de la Figura 5.6.1, especificamos la ruta de la gramática <e-Tutor> definida en la sección 5.4. En la pestaña de *Ejecución de aplicaciones*, insertamos el paquete de imágenes en la ruta de generación de la aplicación, puesto que la instancia XML de la sección 5.3 lo requiere. Por último, especificando la ruta de dicho documento XML y mediante el botón *Ejecutar aplicación*, se lanza como resultado la aplicación <e-Tutor> con el tutorial que se ha definido en el documento XML y que muestra textos e imágenes cada cierto tiempo más un cuadro de texto donde insertar las respuestas a las preguntas que nos realiza.

En la Figura 5.6.2, la ventana de la derecha muestra el ejemplo de ejecución del tutorial definido en el documento XML que se ha seguido en todo el capítulo como base de ejemplo. A continuación, se realiza una demostración de las características avanzadas que ofrece <e-Tutor> mediante la ejecución de un documento XML más complejo.

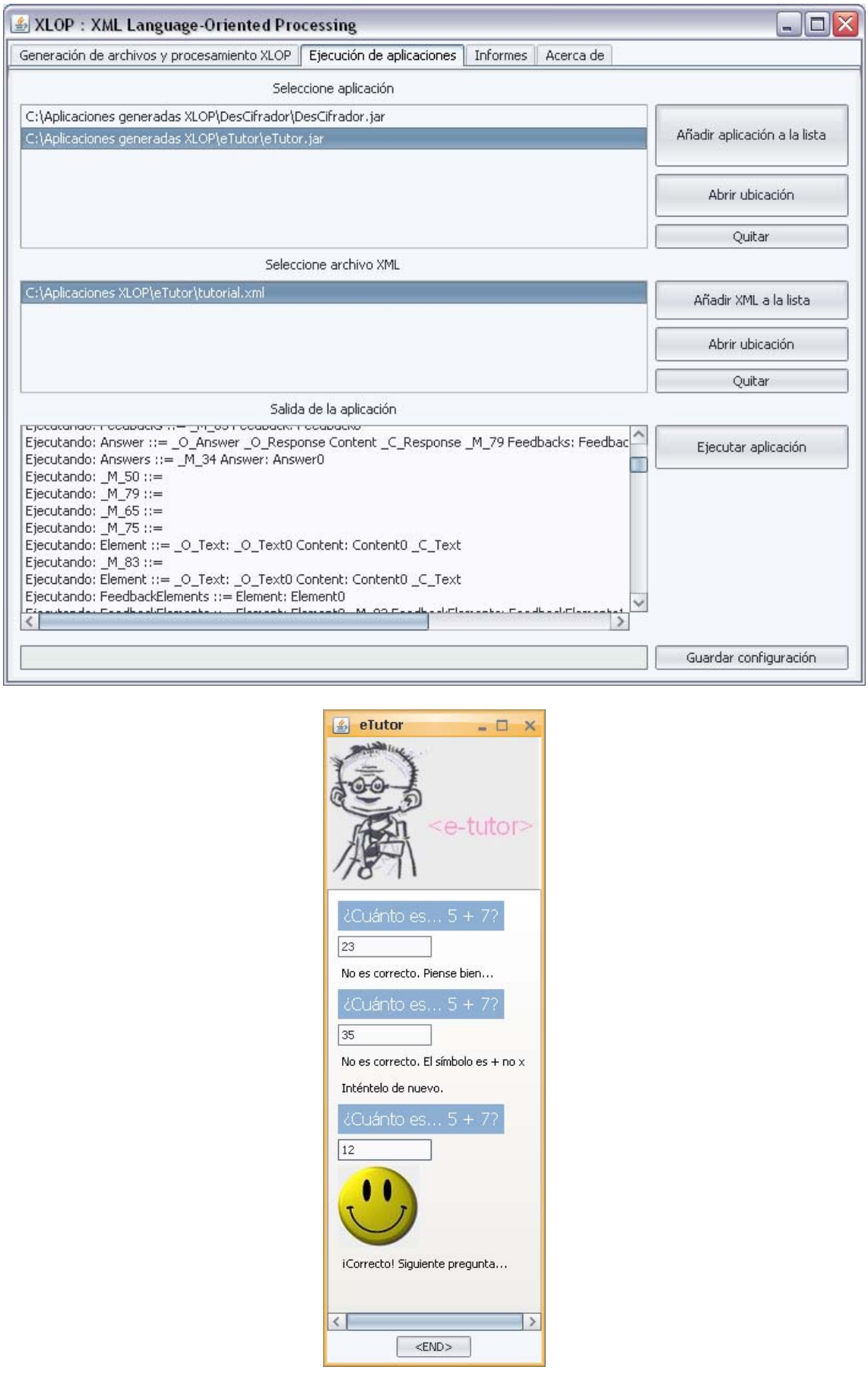

Figura 5.6.2. Ejecución de la aplicación <e-Tutor>.

5.7 Tutorial interactivo: ¡Aprende a conducir!

Las posibilidades que presentaba <e-Tutor> en las secciones anteriores ha ido evolucionando con respecto a su versión inicial permitiendo la posibilidad de incluir elementos multimedia, tales como vídeos desde disco duro o internet (YouTube), música y archivos Flash (.swf). También el usuario ahora es capaz de interactuar con una imagen pulsando en la zona deseada, lo que actuará como posible respuesta a la cuestión planteada en el problema.

Estas mejoras han sido posibles gracias a la facilidad de cambios que ha supuesto el desarrollar la aplicación mediante el entorno XLOP. Para mostrar todas las posibilidades que presenta ahora <e-Tutor>, se ha creado un pequeño pero sofisticado tutorial de demostración que no es más que un test de conducir con elementos interactivos como complemento.

Figura 5.7.1. Pantalla de presentación de <e-Tutor>.

Tras realizar con éxito el análisis de un tutorial o instancia XML, <e-Tutor> permite mostrar una pantalla de presentación creada a partir de un archivo de imagen (Figura 5.7.1). A continuación dará comienzo el tutorial.

La Figura 5.7.2 muestra un ejemplo de secuencia inicial. Pulsando sobre la imagen que muestra el profesor e-Tutor, se carga un video desde YouTube con una cuenta atrás. Tras la finalización del video, se pasa directamente al primer problema. Es posible saltar los vídeos mostrados en un cuadro de diálogo pulsando fuera de él. En el primer problema se pide al usuario que pulse sobre una determinada señal. Si pulsa sobre una incorrecta, se le dará una explicación y se le permitirá intentarlo de nuevo. En caso de acertar la pregunta, el usuario avanza al siguiente problema.

Figura 5.7.2. Tutorial de conducir con <e-Tutor>. Parte I.

Figura 5.7.3. Tutorial de conducir con <e-Tutor>. Parte II.

En la secuencia de problema ilustrada en la Figura 5.7.3, se pide al usuario que introduzca la respuesta correcta en un cuadro de texto. Nótese que esta vez la respuesta del problema es más abierta que en la situación de selección en una imagen que se dio anteriormente. En este caso, el usuario puede introducir texto desde teclado. Sin embargo, la única respuesta válida es la coincidente con el carácter 'A', que corresponde con la respuesta correcta, lo que dará paso al siguiente problema. Otra posible respuesta será interpretada como respuesta incorrecta, pasando a dar la explicación oportuna. Concretamente, la explicación se presenta mediante un contenido Flash interactivo.

Figura 5.7.4. Tutorial de conducir con <e-Tutor>. Parte III.

El siguiente problema de la Figura 5.7.4 da comienzo con un archivo Flash que, a diferencia de otros, no se muestra en un cuadro de diálogo. El conjunto de problemas que vienen a continuación son similares al anterior, a diferencia de que existen respuestas incorrectas que el usuario puede introducir mediante texto, pudiendo dar una explicación concreta según la respuesta errónea contestada.

Por último, la Figura 5.7.5 presenta el último problema de este pequeño tutorial de demostración, que tiene la particularidad de mostrar un video en formato AVI desde el disco duro como explicación a una contestación incorrecta.

Figura 5.7.5. Tutorial de conducir con <e-Tutor>. Parte IV.

Capítulo 6
Conclusiones y trabajo futuro

6.1 Conclusiones

En este trabajo se ha desarrollado un entorno de procesamiento de documentos XML basado en gramáticas de atributos completamente funcional. Se ha mostrado, así mismo, cómo este entorno puede utilizarse en el desarrollo práctico de aplicaciones de procesamiento XML no triviales. Para ello se ha refactorizado <e-Tutor>, un entorno para la generación de sistemas tutoriales a partir de documentos XML. De esta forma, el trabajo ha satisfecho completamente los objetivos planteados al comienzo del mismo. Más concretamente:

- El trabajo nos ha permitido profundizar en el conocimiento de las tecnologías XML, lo que ha resultado un valioso complemento a los conocimientos ya adquiridos sobre el tema en nuestros estudios de Ingeniería Informática. Así mismo, también hemos profundizado en los conocimientos propios de la materia de Procesadores de Lenguaje (y, en particular, en el uso de herramientas de construcción de procesadores –JavaCC y CUP-, así como en el formalismo de las gramáticas de atributos).

- Hemos desarrollado una primera versión completamente funcional de XLOP. Esta versión ha sido (y está siendo) de gran utilidad para mostrar la factibilidad práctica de *ver* las aplicaciones de procesamiento XML como procesadores de lenguaje. XLOP introduce una forma muy sistemática de construir dichas aplicaciones. Efectivamente, las aplicaciones se centran en torno a una especificación declarativa del procesamiento (una gramática de atributos). Esta especificación permite aislar la parte de procesamiento de la lógica específica de la aplicación, lo que fomenta un proceso de construcción racional y sistemático de aplicaciones XML.

- Hemos comprobado la potencialidad de XLOP en el desarrollo (refactorización y extensión, en realidad) de una aplicación de procesamiento XML no trivial: <e-Tutor>. De hecho, <e-Tutor> es, en realidad, un *generador* de aplicaciones. De esta forma, XLOP puede entenderse como un *metagenerador* de aplicaciones, lo que encuentra un paralelismo bastante grande con las modernas tendencias al desarrollo de *software* dirigido por modelos [Stahl et al. 2006]. La diferencia es que, en ciertos dominios de aplicación (como es el caso de las aplicaciones e-Learning tipo tutores <e-Tutor>), las aplicaciones se describen mejor en forma de documentos que en forma de *diagramas* (como proponen los enfoques al desarrollo de software dirigidos por modelos). Esto está en línea con el desarrollo

de software rico en contenidos dirigido por documentos (*paradigma documental*) con el que se ha experimentado (y se está experimentando) en el grupo de e-Learning <e-UCM> [Sierra et al. 2006a; Sierra et al. 2008b].

Por último, el trabajo también nos ha permitido colaborar activamente en un proyecto de investigación, investigación a la que también hemos contribuido activamente a través de la publicación de tres artículos en conferencias internacionales [Sarasa et al. 2009a, 2009d-e]. En este contexto hemos comprobado, además, el valor de aplicar métodos de desarrollo *ágiles* a fin de tratar con la incertidumbre de requisitos incompletos y cambiantes.

6.2 Trabajo futuro

El trabajo iniciado en este proyecto no termina, en absoluto, aquí. Efectivamente, surgen múltiples líneas de trabajo futuro. Estas líneas pueden clasificarse en *líneas a corto*, *medio* y *largo plazo*:

- A corto plazo surge la implementación del algoritmo de marcado refinado, que funcione incluso para *gramáticas patológicas* como la mostrada en el capítulo de desarrollo. Esta implementación podrá realizarse tan pronto el equipo de investigación produzca dicho refinamiento.

- A medio plazo se plantea la refactorización de XLOP, una vez que el proyecto de investigación alcance un estado estable. Entre otros aspectos, puede llevarse a cabo una refactorización del proceso de optimización, acomplando fases a fin de obtener mayor eficiencia. Así mismo, también a medio plazo se plantea el uso de XLOP en el desarrollo de otras aplicaciones (por ejemplo, refactorización de la aplicación de búsqueda de caminos mínimos en redes de metro descrita en [Sierra et al. 2006a], y refactorización del sistema de gestión de contenidos educativos descrito en [Sierra et al. 2006b] mediante el uso de servicios *web*).

- A más largo plazo se plantea la evolución del lenguaje de especificación de XLOP, y del entorno mismo, para soportar características tales como: tipado estático, modularidad, patrones de atribución (p.ej. atributos *remotos* que eviten la proliferación de ecuaciones de copia) [Kastens 1991], etc. También se plantea la creación de un entorno de depuración gráfico, así como de un entorno integrado de desarrollo basado en Eclipse.

Referencias

1. Ablas, H. Attribute Evaluation Methods. En Ablas, H., Melinchar, B (eds.) “Attribute Grammars, Application and Systems”. Lecture Notes in Computer Science 545. Springer. 1991
2. Adobe Systems Inc. Postscript(R) Language Tutorial and Cookbook (APL). Addison-Wesley. 2007
3. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D. Compilers: principles, techniques and tools (second edition). Addison-Wesley. 2007
4. Appel, A.W. Modern Compiler Implementation in Java. Cambridge Univ. Press. 1997
5. Bennett, J.P. Introduction to Compiling Techniques: A First Course using ANSI C, LEX and YACC. McGraw Hill. 1990
6. Birbeck, M et al. Professional XML 2nd Edition. WROX Press, Birminghan,UK. 2001
7. Bork, A. Personal Computers for Education. Harper & Rows. 1985
8. Bradley, N. The XML Companion (3rd Edition). Addison-Wesley. 2001
9. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation. 2008 (disponible en http://www.w3.org/TR/REC-xml/)
10. Brownell, D. SAX2 – Procesing XML Efficiently with Java. O’Relley. 2002
11. Cleaveland, J.C. Program Generators with XML and Java. Prentice-Hall. 2001
12. Coombs, J. H. Renear, A. H., DeRose, S. J. Markup Systems and the Future of Scholarly Text Processing. *Communications of the ACM*, 30 (11), 933-947. 1987
13. DOM. Document Object Model Technical Reports. W3C Recommendations. http://www.w3.org/DOM/DOMTR. 2009
14. Emerson, S. Troff Typesetting for UNIX Systems. Prentice Hall. 1986
15. Gagnon, E. SableCC, an Object-Oriented Compiler Framework. Ph.D. Thesis. School of Computer Science. McGill University, Montreal, 1998
16. Gajardo, L., Mateu, L. Análisis Semántico de Programas Escritos en Java. Theoria, 13, 29-49. 2004
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1994
18. Goldfarb, C. The SGML Handbook. Oxford University Press. 1991
19. Gosling, J., Joy, B., Steele, G.L., Bracha, G. The Java Language Specification Third Edition. Addison Wesley. 2005
20. Grune, D.; Jacobs, C.J.H. Parsing Tecniques – A Practical Guide 2nd Edition. Monographs in Computer Science. Springer. 2008

21. Hudson, S.E. CUP User's Manual. Disponible on-line http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html. 1999

22. Ibrahim, B. Software Engineering Techniques for CAL. Education & Computers 5, 215-222. 1989

23. Kastens, U. Attribute Grammars as an Specification Method. En Ablas, H., Melinchar, B (eds.) "Attribute Grammars, Application and Systems". Lecture Notes in Computer Science 545. Springer. 1991

24. Kawaguchi, K. Flexible Data-Biding with RelaxNGCC. Extreme Markup Languages 2002, August 4-9, Montreal, Canada. 2002

25. Kay, M (ed.). XSL Transformations (XSLT) Version 2.0. W3C Recommendation (disponible en http://www.w3.org/TR/xslt20/). 2007

26. Kleppe, A. Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley. 2008

27. Knuth, D. E. Semantics of Context-free Languages. Mathematical System Theory 2(2), 127–145. Ver también Math. System Theory 5(1), 95–96. 1968.

28. Kodaganallur, V., Incorporating Language Processing into Java Applications: A JavaCC Tutorial, *IEEE Software* 21(4), 70-77, 2004

29. Lam, T.C., Ding, J.J., Liu, J.C. XML Document Parsing: Operational and Performance Characteristics. IEEE Computer 41(9), 30-37. 2008

30. Larman, C. Agile & Iterative Development: A Management Guide. Adisson-Wesley. 2003

31. Maruyama, H. et al. XML and Java: Developing Web Applications. Addison-Wesley. 2002

32. Murata, M., Lee, D., Mani, M., Kawaguchi, K. Taxonomy of XML schema languages using formal language theory. ACM Transactions on Internet Technology 5(4), 660-704. 2005

33. Paaki, J. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Computing Surveys, 27, 2, 196-255. 1995

34. Parr, T. The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf. 2007

35. Purdom, P., Brown, C.A. Semantic Routines and LR(k) parsers. Acta Informatica 14, 299-315. 1980

36. Sarasa, A., Martínez-Avilés, A., Sierra, J.L., Fernández-Valmayor, A. A Generative Approach to the Construction of Application-Specific XML Processing Components. 35th Euromicro Software Engineering and Advanced Applications Conference. 2009a

37. Sarasa, A., Navarro, I., Sierra, J.L, Fernández-Valmayor, A. Building a Syntax Directed Processing Environment for XML Documents by Combining SAX and JavaCC. 3rd International Workshop on XML Data Management Tools & Techniques. DEXA'08. 2008

38. Sarasa, A., Sierra, J.L. Fernández-Valmayor, A. Procesamiento de Documentos XML Dirigido por Lenguajes en Entornos de E-Learning. IEEE RITA, *en prensa*. 2009b

39. Sarasa, A., Sierra, J.L. Fernández-Valmayor, A.Processing Learning Objects with Attribute Grammars. 9th IEEE International Conference on Advanced Learning Technologies. 2009c

40. Sarasa, A., Temprado, B., Sierra, J.L. Fernández-Valmayor, A. XML Language-Oriented Processing with XLOP. 5th International Symposium on Web and Mobile Information Services. 2009d

41. Sarasa, A., Temprado-Battad, B., Martínez-Avilés, A., Sierra, J.L., Fernández-Valmayor, A. Building an Enhanced Syntax-Directed Processing Environment for XML Documents by Combining StAX and CUP. Fourth International Workshop on Flexible Database and Information System Technology. DEXA'09. 2009e

42. Seroul, R., Levy, S., Foata, D. A Beginner's Book of TEX. Springer. 2008

43. Sierra JL, Fernández-Manjón B., Fernández-Valmayor A. A Language-Driven Approach for the Design of Interactive Applications. Interacting with Computers 20(1): 112-127. 2008a

44. Sierra JL, Fernández-Valmayor A., Fernández-Manjón B. How to prototype an educational modeling language. Proceedings of the IX International Simposium on Computers in Education. 2007 November 14-16; Porto, Portugal. pp. 97-102. 2007

45. Sierra, J. L, Fernández-Valmayor, A., Fernández-Manjón, B. A Document-Oriented Paradigm for the Construction of Content-Intensive Applications. The Computer Journal, 49(5), 562-584. 2006a

46. Sierra, J.L., Fernández-Valmayor, A., Fernández-Manjón, B. From Documents to Applications Using Markup Languages. IEEE Software 25(2), 68-76. 2008b

47. Sierra, J.L., Fernández-Valmayor, A., Guinea, M., Hernánz, H. From Research Resources to Virtual Objects: Process model and Virtualization Experiences. J. of Ed. Tech. & Society, 9(3), 56-68. 2006b

48. Sleeman, D., Brown,J.S (eds.). Intelligent Tutoring Systems. Academic Press. 1986

49. Stahl T, Voelter M, Czarnecki K. Model-Driven Software Development, Technology, Engineering, Management. John Wiley & Sons. 2006

50. Stanchfield, S., ANTXR: Easy XML Parsing based on The ANLR Parser Generator. Java Due.com, Hillcrest Comm. & FGM, Inc. javadude.com/tools/antxr/index.html. 2009

51. Vlist, E. Relax NG. O'Relly. 2003

Apéndice A: Componentes en el desarrollo XLOP de <e-Tutor>

A.1. Instancia XML de <e-Tutor>. Tutorial: ¡Aprende a conducir!

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Tutorial SYSTEM "tutorial.dtd">
<Tutorial start="welcome" height="400" width="500"
          title="eTutor - ¡Aprende a conducir!"
          logoPath="imgs/header.png" splashPath="imgs/splashScreen.png"
          color="FFFFFF" color2="FOEEEB" endBtn="FINALIZAR">
      <Features>
            <Feature name="notAnswerErrorMsg">
                  There's not a matching answer
            </Feature>
            <Feature name="notAnswerErrorTitle">ERROR</Feature>
            <Feature name="notFeedbackErrorMsg">
                  There's not feedback
            </Feature>
            <Feature name="notFeedbackErrorTitle">ERROR</Feature>
      </Features>
      <Problem id="welcome">
            <Text fontStyle="1" fontScaleFactor="2" colorf="3399FF">
                  ¡Bienvenido a e-Tutor!
            </Text>
            <Image path="imgs/bocadilloArriba.png"/>
            <Text fontStyle="2" fontScaleFactor="1.1">
                  Aprovechando las ventajas que nos ofrece e-Tutor hemos
                  diseñado un tutorial interactivo que te ayudará a sacarte
                  el carné de conducir.
            </Text>
            <Text fontStyle="2" fontScaleFactor="1.1">
                  Aquí encontrarás una serie de preguntas de conducir,
                  algunas sacadas de diversos exámenes de la DGT. Todas
                  ellas vienen complementadas con explicaciones, animaciones
                  y vídeos para que no se te escape un solo detalle sobre la
                  solución correcta.
            </Text>
            <Text fontStyle="2" fontScaleFactor="1.1">
                  Haz clic sobre mi para comenzar.
            </Text>
            <Image path="imgs/bocadilloAbajo.png"/>
            <ImageQuestionPoint path="imgs/profetransp2.png">
                  <Answer>
                        <Response>
                              37, 69, 16, 62, 3, 53, 5, 43, 17, 46, 8, 27,
                              32, 3, 60, 4, 80, 23, 78, 45, 86, 45, 88, 55,
                              85, 63, 78, 63, 70, 67, 66, 62, 61, 67, 67,
                              77, 73, 75, 74, 94, 84, 125, 7, 124, 26, 88,
                              28, 74, 34, 76
                        </Response>
                        <Feedback next="p1">
                              <YouTube path="TOsGAxFcYls" dcolor="74b8d4"/>
                        </Feedback>
                  </Answer>
                  <AnotherAnswer>
                        <Feedback next="p1">
                              <YouTube path="TOsGAxFcYls" dcolor="74b8d4"/>
                        </Feedback>
                  </AnotherAnswer>
            </ImageQuestionPoint>
      </Problem>
```

Instancia XML del tutorial de conducir. Inicialización y primera pantalla.

```
<Problem id="p1">
	<Text cleanScreen="yes" fontStyle="1" fontScaleFactor="1.5"
		colorf="ffffff" colorb="DD4444">SEÑALES</Text>
	<Text fontStyle="1" fontScaleFactor="1.2">
		¿Cuál de las siguientes señales indica...
	</Text>
	<Text fontStyle="2" colorf="006699">
		Peligro por la presencia inmediata de un paso a nivel sin
		barreras con más de una vía férrea
	</Text>
	<ImageQuestionPoint path="imgs/señales.png">
		<Answer>
			<Response>48, 66, 47, 7, 28, 7, 28, 66</Response>
			<Feedback next="p1">
				<Image delay="2000" path="imgs/semaforoNO.gif"/>
				<Text delay="4000" colorf="FF0000"
					fontScaleFactor="1.2">
					Indica, en el lado izquierdo, la aproximación
					a un paso a nivel, puente móvil o muelle, que
					dista de éste al menos dos tercios de la
					distancia entre él y la correspondiente señal
					de advertencia del peligro.
				</Text>
			</Feedback>
			<Feedback next="p2">
				<Image delay="2000" path="imgs/semaforoNO.gif"/>
				<Text colorf="FF0000" fontScaleFactor="1.2">
					La señal es esta
				</Text>
				<Image delay="3000" path="imgs/P-11A.gif"/>
			</Feedback>
		</Answer>
		<Answer>
			<Response>
				113, 40, 89, 55, 83, 53, 84, 46, 104, 34, 83, 23,
				83, 19, 89, 16, 113, 30, 138, 16, 144, 18, 143, 24,
				122, 35, 142, 47, 145, 53, 139, 54
			</Response>
			<Feedback next="p1">
				<Image delay="2000" path="imgs/semaforoNO.gif"/>
				<Text delay="4000" colorf="FF0000"
					fontScaleFactor="1.2">
					Situación de un paso a nivel sin barreras.
					Peligro por la presencia inmediata de un paso
					a nivel sin barreras.
				</Text>
			</Feedback>
			<Feedback next="p2">
				<Image delay="2000" path="imgs/semaforoNO.gif"/>
				<Text colorf="FF0000" fontScaleFactor="1.2">
					La señal es esta</Text>
				<Image delay="3000" path="imgs/P-11A.gif"/>
			</Feedback>
		</Answer>
		<Answer>
			<Response>37, 80, 69, 135, 6, 135</Response>
			<Feedback next="p1">
				<Image delay="2000" path="imgs/semaforoNO.gif"/>
				<Text delay="4000" colorf="FF0000"
					fontScaleFactor="1.2">
					Peligro ante la proximidad de un puente que
					puede ser levantado o girado,
					interrumpiéndose así temporalmente la
					circulación.
				</Text>
			</Feedback>
			<Feedback next="p2">
				<Image delay="2000" path="imgs/semaforoNO.gif"/>
				<Text colorf="FF0000" fontScaleFactor="1.2">
					La señal es esta
				</Text>
				<Image delay="3000" path="imgs/P-11A.gif"/>
			</Feedback>
		</Answer>
```

Instancia XML del tutorial de conducir. Primer problema (I).

```
            <Answer>
                  <Response>83, 136, 113, 80, 145, 135</Response>
                  <Feedback next="p1">
                        <Image delay="2000" path="imgs/semaforoNO.gif"/>
                        <Text delay="4000" colorf="FF0000"
                              fontScaleFactor="1.2">
                              Peligro por la proximidad de un paso a nivel
                              provisto de barreras o semibarreras.
                        </Text>
                  </Feedback>
                  <Feedback next="p2">
                        <Image delay="2000" path="imgs/semaforoNO.gif"/>
                        <Text colorf="FF0000" fontScaleFactor="1.2">
                              La señal es esta
                        </Text>
                        <Image delay="3000" path="imgs/P-11A.gif"/>
                  </Feedback>
            </Answer>
            <Answer>
                  <Response>159, 136, 188, 78, 221, 135</Response>
                  <Feedback next="p1">
                        <Image delay="2000" path="imgs/semaforoNO.gif"/>
                        <Text delay="4000" colorf="FF0000"
                              fontScaleFactor="1.2">
                              Peligro por la proximidad de un paso a nivel
                              no provisto de barreras o
                              semibarreras.
                        </Text>
                  </Feedback>
                  <Feedback next="p2">
                        <Image delay="2000" path="imgs/semaforoNO.gif"/>
                        <Text colorf="FF0000" fontScaleFactor="1.2">
                              La señal es esta
                        </Text>
                        <Image delay="3000" path="imgs/P-11A.gif"/>
                  </Feedback>
            </Answer>
            <Answer>
                  <Response>
                        163, 67, 190, 51, 214, 65, 221, 62, 219, 55, 189,
                        41, 191, 32, 213, 45, 220, 43, 218, 37, 198, 25,
                        219, 13, 219, 8, 214, 7, 190, 21, 164, 8, 158, 9,
                        160, 15, 179, 26, 159, 37, 158, 46, 164, 47, 187,
                        32, 186, 43, 162, 57, 159, 63
                  </Response>
                  <Feedback next="p2">
                        <Image delay="2000" path="imgs/semaforoYES.gif"/>
                        <Text delay="4000">
                              Pasemos a la siguiente pregunta.
                        </Text>
                  </Feedback>
            </Answer>
            <AnotherAnswer>
                  <Feedback next="p1">
                        <Text delay="2000">
                              ¡Tienes que pinchar en una señal!.
                        </Text>
                  </Feedback>
                  <Feedback next="p2">
                        <Text delay="2000">
                              Has pinchado donde no debias, te quedas sin
                              saber la respuesta.
                        </Text>
                  </Feedback>
            </AnotherAnswer>
      </ImageQuestionPoint>
</Problem>
```

Instancia XML del tutorial de conducir. Primer problema (II).

```
<Problem id="p2">
	<Text cleanScreen="yes" fontStyle="1" fontScaleFactor="1.5"
		colorf="ffffff" colorb="DD4444">SEÑALES</Text>
	<Text fontStyle="1" fontScaleFactor="1.2">
		Los dispositivos de guía...
	</Text>
	<Image delay="1500" path="imgs/dispositivoguia.gif"/>
	<Text delay="1000" colorf="006699">
		A. son señales de balizamiento fijo.
	</Text>
	<Text delay="1000" colorf="006699">
		B. son dispositivos de barrera.
	</Text>
	<Text delay="1000" colorf="006699">
		C. Ambas respuestas son falsas.
	</Text>
	<QuestionPoint>
		<Answer>
			<Response>A</Response>
			<Feedback next="p3">
				<Image delay="2000" path="imgs/semaforoYES.gif"/>
				<Text delay="2000">
					Pasemos a la siguiente pregunta.
				</Text>
			</Feedback>
		</Answer>
		<AnotherAnswer>
			<Feedback next="p2">
				<Image delay="2000" path="imgs/semaforoNO.gif"/>
				<Text delay="3000" colorf="FF0000"
					fontScaleFactor="1.2">
					Observa la siguiente animación e inténtalo de
					nuevo.
				</Text>
				<Flash path="imgs\balizamiento.swf" height="320"
					width="400"/>
			</Feedback>
			<Feedback next="p3">
				<Image delay="2000" path="imgs/semaforoNO.gif"/>
				<Text delay="2000" colorf="FF0000"
					fontScaleFactor="1.2">
					La respuesta correcta es la A.
				</Text>
			</Feedback>
		</AnotherAnswer>
	</QuestionPoint>
</Problem>
```

Instancia XML del tutorial de conducir. Segundo problema.

```
<Problem id="p3">
	<Text cleanScreen="yes" fontStyle="1" fontScaleFactor="1.5"
	      colorf="ffffff" colorb="44DD44">PANEL DE MANDOS</Text>
	<Text>
		Pasando el ratón por los diferentes relojes y luminosos de este
		panel de mandos podrás encontrar una breve descripción de la
		utilidad de los mismos.
	</Text>
	<Flash path="imgs\mandos.swf" dialog="no" height="295" width="460"/>
	<Text cleanScreen="yes" fontStyle="1" fontScaleFactor="1.5"
	      colorf="ffffff" colorb="44DD44">PANEL DE MANDOS</Text>
	<Text>Responde ahora a las siguientes preguntas:</Text>
	<Text fontStyle="1" fontScaleFactor="1.2">
		Si la luz del testigo de carga del generador permanece apagada
		durante la marcha, ocurre que...
	</Text>
	<Image delay="1500" path="imgs/testigocarga.jpg"/>
	<Text delay="1000" colorf="006699">
		A. el generador produce energía eléctrica correctamente.</Text>
	<Text delay="1000" colorf="006699">
		B. el generador está desconectado.
	</Text>
	<Text delay="1000" colorf="006699">
		C. el amperímetro no marca ni carga ni descarga.
	</Text>
	<QuestionPoint>
		<Answer>
			<Response>A</Response>
			<Feedback next="p3b">
				<Image delay="2000" path="imgs/semaforoYES.gif"/>
				<Text delay="2000">
					Pasemos a la siguiente pregunta.
				</Text>
			</Feedback>
		</Answer>
		<AnotherAnswer>
			<Feedback next="p3b">
				<Image delay="2000" path="imgs/semaforoNO.gif"/>
				<Text delay="4000" colorf="FF0000"
				      fontScaleFactor="1.2">
					La respuesta correcta era la A. Recuerda que
					se trata de una alerta que solo se enciende
					en caso de que el circuito de carga no
					funcione correctamente.
				</Text>
			</Feedback>
		</AnotherAnswer>
	</QuestionPoint>
</Problem>

<Problem id="p3b">
	<Text cleanScreen="yes" fontStyle="1" fontScaleFactor="1.5"
	      colorf="ffffff" colorb="44DD44">PANEL DE MANDOS</Text>
	<Text fontStyle="1" fontScaleFactor="1.2">
		Si se enciende la luz roja del indicador de la presión de
		lubricación del salpicadero, sabrás que la lubricación
		es...
	</Text>
	<Text delay="1000" colorf="006699">
		A. imperfecta; la presión de lubricación es muy alta.
	</Text>
	<Text delay="1000" colorf="006699">
		B. imperfecta; la temperatura del aceite es muy baja.
	 </Text>
		<Text delay="1000" colorf="006699">
		C. imperfecta; la presión de lubricación es más baja que la
		necesaria.
	</Text>
```

Instancia XML del tutorial de conducir. Tercer problema (A y B).

```
        <QuestionPoint>
                <Answer>
                        <Response>A</Response>
                        <Feedback next="p3b">
                                <Image delay="2000" path="imgs/semaforoNO.gif"/>
                                <Text colorf="FF0000" fontScaleFactor="1.2">
                                        Recuerda que el testigo de presión tiene el
                                        siguiente aspecto. Inténtalo de nuevo.
                                </Text>
                                <Image delay="4000" path="imgs/testigopresion.jpg"/>
                        </Feedback>
                        <Feedback next="p3c">
                                <Image delay="2000" path="imgs/semaforoNO.gif"/>
                                <Text delay="4000" colorf="FF0000"
                                        fontScaleFactor="1.2">
                                        La respuesta correcta es la C. Pasemos a la
                                        siguiente pregunta.
                                </Text>
                        </Feedback>
                </Answer>
                <Answer>
                        <Response>B</Response>
                        <Feedback next="p3b">
                                <Image delay="2000" path="imgs/semaforoNO.gif"/>
                                <Text delay="4000" colorf="FF0000"
                                        fontScaleFactor="1.2">
                                        Este indicador mide la presión, no la
                                        temperatura. Prueba de nuevo.
                                </Text>
                        </Feedback>
                        <Feedback next="p3c">
                                <Image delay="2000" path="imgs/semaforoNO.gif"/>
                                <Text delay="4000" colorf="FF0000"
                                        fontScaleFactor="1.2">
                                        La respuesta correcta es la C. Pasemos a la
                                        siguiente pregunta.
                                </Text>
                        </Feedback>
                </Answer>
                <Answer>
                        <Response>C</Response>
                        <Feedback next="p3c">
                                <Image delay="2000" path="imgs/semaforoYES.gif"/>
                                <Text delay="2000">
                                        Pasemos a la siguiente pregunta.
                                </Text>
                        </Feedback>
                </Answer>
                <AnotherAnswer>
                        <Feedback next="p3b">
                                <Text delay="2000">
                                        Por favor, contesta A, B, o C. Asegúrate que
                                        sean letras mayúsculas
                                </Text>
                        </Feedback>
                        <Feedback next="p3c">
                                <Text delay="2000">
                                        Demasiado difícil para tí. Prueba con el
                                        siguiente vídeo.
                                </Text>
                                <YouTube path="0gssaVuvKwk" dcolor="74b8d4"/>
                        </Feedback>
                </AnotherAnswer>
        </QuestionPoint>
</Problem>
```

Instancia XML del tutorial de conducir. Tercer problema (B cont.).

```
<Problem id="p3c">
	<Text cleanScreen="yes" fontStyle="1" fontScaleFactor="1.5"
		colorf="ffffff" colorb="44DD44">PANEL DE MANDOS</Text>
	<Text fontStyle="1" fontScaleFactor="1.2">
		Si, al accionar el indicador de dirección, la lámpara testigo
		parpadea más rápido de lo normal, es un indicio de que...
	</Text>
	<Image delay="1500" path="imgs/indicadordireccion.gif"/>
	<Text delay="1000" colorf="006699">
		A. alguna lámpara de los indicadores no luce.
	</Text>
	<Text delay="1000" colorf="006699">
		B. existe un cortocircuito.
	</Text>
	<Text delay="1000" colorf="006699">
		C. están encendidos todos los indicadores de dirección.
	</Text>
	<QuestionPoint>
		<Answer>
			<Response>A</Response>
			<Feedback next="p4">
				<Image delay="2000" path="imgs/semaforoYES.gif"/>
				<Text delay="2000">
					Pasemos a la siguiente pregunta.
				</Text>
			</Feedback>
		</Answer>
		<Answer>
			<Response>B</Response>
			<Feedback next="p4">
				<Image delay="2000" path="imgs/semaforoNO.gif"/>
				<Text delay="5000" colorf="FF0000"
					fontScaleFactor="1.2">
					La respuesta correcta es la A. Si el testigo
					luminoso parpadea más rapido de lo normal,
					puede ser porque se haya fundido la lámpara
					de alguno de los intermitentes.
				</Text>
			</Feedback>
		</Answer>
		<Answer>
			<Response>C</Response>
			<Feedback next="p4">
				<Image delay="2000" path="imgs/semaforoNO.gif"/>
				<Text delay="4000" colorf="FF0000"
					fontScaleFactor="1.2">
					La respuesta correcta es la A. Si todos los
					indicadores de dirección se encuentran
					encendidos ámbos testigos se encontrarían
					parpadeando a un ritmo normal.
				</Text>
			</Feedback>
		</Answer>
		<AnotherAnswer>
			<Feedback next="p3c">
				<Text delay="2000">
					Por favor, contesta A, B, o C. Asegúrate que
					sean letras mayúsculas
				</Text>
			</Feedback>
			<Feedback next="p4">
				<Text delay="2000">
					Demasiado difícil para tí. Prueba con el
					siguiente vídeo.
				</Text>
				<YouTube path="0gssaVuvKwk" dcolor="74b8d4"/>
			</Feedback>
		</AnotherAnswer>
	</QuestionPoint>
</Problem>
```

Instancia XML del tutorial de conducir. Tercer problema (C).

```
	<Problem id="p4">
		<Text cleanScreen="yes" fontStyle="1" fontScaleFactor="1.5"
		      colorf="ffffff" colorb="4444DD">MECÁNICA</Text>
		<Text fontStyle="1" fontScaleFactor="1.2">
		      ¿Cuál es el elemento frenante en los frenos de
		      tambor?
		</Text>
		<Text delay="1000" colorf="006699"> A. Los discos. </Text>
		<Text delay="1000" colorf="006699"> B. Las zapatas. </Text>
		<Text delay="1000" colorf="006699"> C. Las llantas. </Text>
		<QuestionPoint>
			<Answer>
				<Response>B</Response>
				<Feedback next="end">
					<Image delay="2000"
					       path="imgs/semaforoYES.gif"/>
				</Feedback>
			</Answer>
			<AnotherAnswer>
				<Feedback next="p4">
					<Image delay="2000"
					       path="imgs/semaforoNO.gif"/>
					<Text delay="1000" colorf="FF0000"
					      fontScaleFactor="1.2">
					      Observa el siguiente video e inténtalo
					      de nuevo.
					</Text>
					<Multimedia path="imgs/v16.avi" height="200"
					            width="300" controlBar="no"/>
				</Feedback>
				<Feedback next="end">
					<Image delay="2000"
					       path="imgs/semaforoNO.gif"/>
					<Text delay="3000" colorf="FF0000"
					      fontScaleFactor="1.2">
					      La respuesta correcta es la B.
					</Text>
				</Feedback>
			</AnotherAnswer>
		</QuestionPoint>
	</Problem>

	<Problem id="end">
		<Image cleanScreen="yes" path="imgs/bocadilloArriba.png"/>
		<Text fontStyle="1" fontScaleFactor="3" colorf="0000FF">
		      ¡FIN de la demostración!
		</Text>
		<Image path="imgs/bocadilloAbajo.png"/>
		<Image path="imgs/profetransp2.png"/>
	</Problem>

</Tutorial>
```

Instancia XML del tutorial de conducir. Cuarto problema y pantalla final.

A.2. DTD completa de <e-Tutor>

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % clrScr "cleanScreen (yes|no) #IMPLIED">
<!ENTITY % cBtn "continueBtn (yes|no) #IMPLIED">
<!ENTITY % sBar "controlBar (yes|no) #IMPLIED">
<!ENTITY % dg "dialog (yes|no) #IMPLIED
               dcolor CDATA #IMPLIED">
<!ENTITY % presentAttrs "title CDATA #REQUIRED
                         endBtn CDATA #IMPLIED
                         logoPath CDATA #IMPLIED
                         splashPath CDATA #IMPLIED
                         color CDATA #IMPLIED
                         color2 CDATA #IMPLIED
                         height CDATA #REQUIRED
                         width CDATA #REQUIRED">
<!ENTITY % textAttrs “colorf CDATA #IMPLIED
                      colorb CDATA #IMPLIED
                      fontStyle CDATA #IMPLIED
                      fontScaleFactor CDATA #IMPLIED">
<!ELEMENT Tutorial (Features?,Problem+)>
<!ATTLIST Tutorial start IDREF #REQUIRED %presentAttrs;>
<!ELEMENT Problem ((QuestionPoint | ImageQuestionPoint) |
((Text|Image|Multimedia|Flash|YouTube)+,(QuestionPoint|ImageQuestionPoint)?))>
<!ATTLIST   Problem id ID #REQUIRED>
<!ELEMENT Text (#PCDATA)>
<!ATTLIST Text delay NMTOKEN "1" %textAttrs; %clrScr;>
<!ELEMENT Image (#PCDATA)>
<!ATTLIST Image path CDATA #REQUIRED delay NMTOKEN "1" %clrScr;>
<!ELEMENT Multimedia (#PCDATA)>
<!ATTLIST Multimedia path CDATA #REQUIRED
                     height CDATA #REQUIRED
                     width CDATA #REQUIRED
                     %dg;
                     %cBtn;
                     %sBar;
                     %clrScr;>
<!ELEMENT Flash EMPTY>
<!ATTLIST Flash path CDATA #REQUIRED
                height CDATA #IMPLIED
                width CDATA #IMPLIED
                %dg;
                %cBtn;
                %clrScr;>
<!ELEMENT YouTube EMPTY>
<!ATTLIST YouTube path CDATA #REQUIRED
                  %dg;
                  %cBtn;
                  %clrScr;>
<!ELEMENT QuestionPoint (Answer+,AnotherAnswer)>
<!ATTLIST QuestionPoint inputColumns NMTOKEN "10">
<!ELEMENT ImageQuestionPoint (Answer+,AnotherAnswer)>
<!ATTLIST ImageQuestionPoint path CDATA #REQUIRED>
<!ELEMENT Feature (#PCDATA)>
<!ATTLIST Feature name CDATA #REQUIRED>
<!ELEMENT Answer (Response, Feedback+)>
<!ELEMENT AnotherAnswer (Feedback+)>
<!ELEMENT Response (#PCDATA)>
<!ELEMENT RespText (#PCDATA)>
<!ELEMENT Feedback ((Text|Image|Multimedia|YouTube|Flash)+)>
<!ATTLIST Feedback next IDREF #IMPLIED>
<!ELEMENT Features (Feature)*>
```

DTD actual para las instancias XML <e-Tutor>.

A.3. Gramática XLOP de <e-Tutor>

```
Tutorial ::= <Tutorial> <Features> Features </Features> Problems </Tutorial> {
   tutorialCreatedh of Problems = after(
                                  featuresStored of Features,
                                  newTutorial(height of <Tutorial>,
                                  width of <Tutorial>,
                                  title of <Tutorial>,
                                  endBtn of <Tutorial>,
                                  newColor(color of <Tutorial>),
                                  newColor(color2 of <Tutorial>),
                                  logoPath of <Tutorial>,
                                  splashPath of <Tutorial>))
   tutorialExecuted of Tutorial = after(
                                  problemsCreated of Problems,
                                  run(getProblem(start of <Tutorial>)))
}

Features ::= Features Feature {
   featuresStored of Features(0) = after(
                                  featuresStored of Features(1),
                                  featureStored of Feature)
}

Features ::= Feature {
  featuresStored of Features = featureStored of Feature
}

Feature ::= <Feature> #pcdata </Feature> {
   featureStored of Feature = addFeature(name of <Feature>, text of #pcdata)
}

Problems ::= Problems Problem {
   tutorialCreatedh of Problems(1) = tutorialCreatedh of Problems(0)
   tutorialCreatedh of Problem = tutorialCreatedh of Problems(0)
   problemsCreated of Problems(0) = after(
                                  problemsCreated of Problems(1),
                                   newProblem(id of Problem, elem of Problem))
}

Problems ::= Problem {
   tutorialCreatedh of Problem = tutorialCreatedh of Problems
   problemsCreated of Problems = after(
                                  tutorialCreatedh of Problems,
                                   newProblem(id of Problem, elem of Problem))
}

Problem ::= <Problem> ProblemDescription </Problem> {
    tutorialCreatedh of ProblemDescription = tutorialCreatedh of Problem
    id of Problem = id of <Problem>
    elem of Problem = elem of ProblemDescription
}

ProblemDescription ::= QuestionPoint {
    tutorialCreatedh of QuestionPoint = tutorialCreatedh of ProblemDescription
    elem of ProblemDescription = qp of QuestionPoint
}
```

Gramática XLOP para <e-Tutor>. Parte I.

```
ProblemDescription ::= Element RestOfProblemDescription {
    tutorialCreatedh of Element = tutorialCreatedh of ProblemDescription
    tutorialCreatedh of RestOfProblemDescription = tutorialCreatedh of
                                                    ProblemDescription
    elem of ProblemDescription = putNext(
                                elem of Element,
                                elem of RestOfProblemDescription)
}

RestOfProblemDescription ::= Element RestOfProblemDescription {
    tutorialCreatedh of Element = tutorialCreatedh of
                                    RestOfProblemDescription(0)
    tutorialCreatedh of RestOfProblemDescription(1) = tutorialCreatedh of
                                    RestOfProblemDescription(0)
    elem of RestOfProblemDescription(0) = putNext(
                                elem of Element,
                                elem of RestOfProblemDescription(1))
}

RestOfProblemDescription ::= QuestionPoint {
    tutorialCreatedh of QuestionPoint = tutorialCreatedh of
                                            RestOfProblemDescription
    elem of RestOfProblemDescription = qp of QuestionPoint
}

RestOfProblemDescription ::= {
    elem of RestOfProblemDescription = voidElem()
}

Element ::= <Text> #pcdata </Text> {
    elem of Element = after(
                            tutorialCreatedh of Element,
                            newText(
                                    text of #pcdata,
                                    delay of <Text>,
                                    newColor(colorf of <Text>),
                                    newColor(colorb of <Text>),
                                    fontStyle of <Text>,
                                    fontScaleFactor of <Text>,
                                    cleanScreen of <Text>))
}

Element ::= <Image/> {
    elem of Element = after(
                            tutorialCreatedh of Element,
                            newImage(
                                    path of <Image>,
                                    delay of <Image>,
                                    cleanScreen of <Image>))
}

Element ::= <Multimedia/> {
   elem of Element = after(
                            tutorialCreatedh of Element,
                            newMultimedia(
                                    path of <Multimedia>,
                                    height of <Multimedia>,
                                    width of <Multimedia>,
                                    dialog of <Multimedia>,
                                    newColor(dcolor of <Multimedia>),
                                    continueBtn of <Multimedia>,
                                    controlBar of <Multimedia>,
                                    cleanScreen of <Multimedia>))
}
```

```
Element ::= <Flash/> {
   elem of Element = after(
                           tutorialCreatedh of Element,
                           newFlash(
                                  path of <Flash>,
                                  height of <Flash>,
                                  width of <Flash>,
                                  dialog of <Flash>,
                                  newColor(dcolor of <Flash>),
                                  continueBtn of <Flash>,
                                  cleanScreen of <Flash>))
}

Element ::= <YouTube/> {
   elem of Element = after(
                           tutorialCreatedh of Element,
                           newYouTube(
                                  path of <YouTube>, dialog of <YouTube>,
                                  newColor(dcolor of <YouTube>),
                                  continueBtn of <YouTube>,
                                  cleanScreen of <YouTube>))
}

QuestionPoint ::= <QuestionPoint> Answers AnotherAnswer </QuestionPoint> {
   tutorialCreatedh of Answers = tutorialCreatedh of QuestionPoint
   tutorialCreatedh of AnotherAnswer = tutorialCreatedh of QuestionPoint
   qph of Answers = after(
                           tutorialCreatedh of QuestionPoint,
                           newQuestion(inputColumnsh of <QuestionPoint>))
   qp of QuestionPoint = addAnswer(qp of Answers, answ of AnotherAnswer)
}

QuestionPoint ::=  <ImageQuestionPoint> Answers AnotherAnswer
</ImageQuestionPoint> {
   tutorialCreatedh of Answers = tutorialCreatedh of QuestionPoint
   tutorialCreatedh of AnotherAnswer = tutorialCreatedh of QuestionPoint
   qph of Answers = after(
                           tutorialCreatedh of QuestionPoint,
                           newImageQuestion(path of <ImageQuestionPoint>))
   qp of QuestionPoint = addAnswer(qp of Answers, answ of AnotherAnswer)
}

Answers ::= Answers Answer {
   tutorialCreatedh of Answers(1) = tutorialCreatedh of Answers(0)
   tutorialCreatedh of Answer = tutorialCreatedh of Answers(0)
   qph of Answers(1) = qph of Answers(0)
   qp of Answers(0) = addAnswer(qp of Answers(1), answ of Answer)
}

Answers ::= Answer {
  tutorialCreatedh of Answer = tutorialCreatedh of Answers
  qp of Answers = addAnswer(qph of Answers, answ of Answer)
}

Answer ::= <Answer> <Response> #pcdata </Response> Feedbacks </Answer> {
   tutorialCreatedh of Feedbacks = tutorialCreatedh of Answer
   answh of Feedbacks = after(
                           tutorialCreatedh of Answer,
                           newAnswer(text of #pcdata))
   answ of Answer = answ of Feedbacks
}
```

Gramática XLOP para <e-Tutor>. Parte III.

A.4. La Clase Semántica de <e-Tutor>

```
import etutor.framework.*;
import java.awt.Color;
import java.awt.Polygon;
import java.util.Hashtable;
import java.util.Stack;

public class TutorialSemanticClass {

   private Hashtable<String,String> features;
   private Hashtable<String,ETTutorialElement> problems;
   private Hashtable<String,Stack<ETTutorialElement>> pendingElements;
   private ETTutorial tutorial;

   public TutorialSemanticClass() {
      features = new Hashtable<String,String>();
      problems = new Hashtable<String,ETTutorialElement>();
      pendingElements = new Hashtable<String,Stack<ETTutorialElement>>();
   }

   public boolean newTutorial(String height, String width, String title,
                              String endBtn, Color color, Color color2,
                              String logoPath, String splashPath) {
      tutorial = new ETTutorial(
       Integer.valueOf(width).intValue(),
       Integer.valueOf(height).intValue());
      tutorial.setTitle(title);
      if (endBtn != null) tutorial.setLabelEndButton(endBtn);
      if (color != null) tutorial.setBlackboardColor(color);
      if (color2 != null) tutorial.setBlackboardColor2(color2);
      if (logoPath != null) tutorial.setLogoLocation(logoPath);
      if (splashPath != null) tutorial.setSplashLocation(splashPath);
      return true;
   }

   public boolean addFeature(String name, String text) {
      features.put(name,text);
      return true;
   }

   public Object after(Object a, Object b) {return b;}

   public void run(ETTutorialElement e) {
      features = null; problems = null; pendingElements = null; System.gc();
      tutorial.setInitialTutorialElement(e);
      tutorial.run();
   }

   public ETTutorialElement newText(String text, String delay,
                                    Color foreground, Color background,
                                    String fontStyle, String fontScaleFactor,
                                    String cleanScreen) {
      ETText etext = new ETText();
      etext.setText(text);
      if (delay != null) etext.setDelay(Integer.valueOf(delay).intValue());
      else etext.setDelay(100);
      if (foreground != null) etext.setForegroundColor(foreground);
      if (background != null) etext.setBackgroundColor(background);
      if (fontStyle != null) etext.setFontStyle(
            Integer.valueOf(fontStyle).intValue());
      if (fontScaleFactor != null) etext.setFontScaleFactor(
            Double.valueOf(fontScaleFactor).doubleValue());
      if (cleanScreen != null) etext.cleanScreen(cleanScreen.equals("yes"));
      etext.setTutorial(tutorial);
      return etext;
   }
```

La clase semántica de <e-Tutor>. Parte I.

```
public ETTutorialElement newImage(String path, String delay, String
                                  cleanScreen) {
   ETImage eimg = new ETImage();
   eimg.setLocation(path);
   eimg.setTutorial(tutorial);
   if (delay != null) eimg.setDelay(Integer.valueOf(delay).intValue());
   else eimg.setDelay(100);
   if (cleanScreen != null) eimg.cleanScreen(cleanScreen.equals("yes"));
   return eimg;
}

public ETTutorialElement newMultimedia(String path, String height,
                                       String width, String dialog,
                                       Color dcolor, String continueBtn,
                                       String controlBar,
                                       String cleanScreen) {
   ETMultimedia m = new ETMultimedia();
   m.setLocation(path);
   if (height != null) m.setHeight(Integer.valueOf(height).intValue());
   if (width != null) m.setWidth(Integer.valueOf(width).intValue());
   if (dialog != null) m.setDialog(dialog.equals("yes"));
   if (dcolor != null) m.setColor(dcolor);
   if (continueBtn != null) m.setContinueBtn(continueBtn.equals("yes"));
   if (controlBar != null) m.setControlBarVisible(
         controlBar.equals("yes"));
   if (cleanScreen != null) m.cleanScreen(cleanScreen.equals("yes"));
   m.setTutorial(tutorial);
   return m;
}

public ETTutorialElement newFlash(String path, String height, String width,
                                  String dialog,
                                  Color dcolor, String continueBtn,
                                  String cleanScreen) {
   ETFlash f = new ETFlash();
   f.setLocation(path);
   if (height != null) f.setHeight(Integer.valueOf(height).intValue());
   if (width != null) f.setWidth(Integer.valueOf(width).intValue());
   if (dialog != null) f.setDialog(dialog.equals("yes"));
   if (dcolor != null) f.setColor(dcolor);
   if (continueBtn != null) f.setContinueBtn(continueBtn.equals("yes"));
   if (cleanScreen != null) f.cleanScreen(cleanScreen.equals("yes"));
   f.setTutorial(tutorial);
   return f;
}

public ETTutorialElement newYouTube(String path, String dialog,
                                    Color dcolor, String continueBtn,
                                    String cleanScreen) {
   ETYouTube y = new ETYouTube();
   y.setLocation(path);
   if (dialog != null) y.setDialog(dialog.equals("yes"));
   if (dcolor != null) y.setColor(dcolor);
   if (continueBtn != null) y.setContinueBtn(continueBtn.equals("yes"));
   if (cleanScreen != null) y.cleanScreen(cleanScreen.equals("yes"));
   y.setTutorial(tutorial);
   return y;
}

public ETTutorialElement getProblem(String id) {
   return problems.get(id);
}

public ETTutorialElement putNext(ETTutorialElement e,
                                 ETTutorialElement ne) {
   e.setNextElement(ne);
   return e;
}
```

La clase semántica de <e-Tutor>. Parte II.

```
public ETTutorialElement voidElem() {return null;}

public ETTutorialElement linkWithProblem(ETTutorialElement e,
                                         String problem) {
   if (problem == null) {  //Fin de tutorial.
      e.setNextElement(null);
      return e;
   }
   ETTutorialElement ne = problems.get(problem);
   if (ne != null) e.setNextElement(ne);
   else {
      Stack<ETTutorialElement> pendingElems = pendingElements.get(problem);
      if (pendingElems == null) {
         pendingElems = new Stack<ETTutorialElement>();
         pendingElements.put(problem,pendingElems);
      }
      pendingElems.push(e);
   }
   return e;
}

public boolean newProblem(String id, ETTutorialElement e) {
   Stack<ETTutorialElement> pendingElms = pendingElements.get(id);
   if (pendingElms != null)
      while(!pendingElms.empty())
         pendingElms.pop().setNextElement(e);
   problems.put(id,e);
   return true;
}

public ETQuestionPoint newQuestion(String inputColumns) {
   ETQuestionPoint qp = new ETQuestionPoint();
   if (inputColumns == null) inputColumns = "10";
   TextInput ti = new TextInput(Integer.valueOf(inputColumns).intValue());
   qp.setInput(ti);
   ti.setInputContinuation(qp);
   qp.setMsgNotAnswer(features.get("notAnswerErrorMsg"));
   qp.setMsgNotAnswerTitle(features.get("notAnswerErrorTitle"));
   return qp;
}

public ETImageQuestionPoint newImageQuestion(String path) {
   ETImageQuestionPoint iqp = new ETImageQuestionPoint();
   ImageInput ii = new ImageInput(path);
   iqp.setInput(ii);
   ii.setInputContinuation(iqp);
   iqp.setMsgNotAnswer(features.get("notAnswerErrorMsg"));
   iqp.setMsgNotAnswerTitle(features.get("notAnswerErrorTitle"));
   return iqp;
}

public Polygon newPolygon() {
   return new Polygon();
}

public Polygon addPoint(Polygon p, String x, String y) {
   p.addPoint(Integer.valueOf(x).intValue(),Integer.valueOf(y).intValue());
   return p;
}

public ETAbstractAnswer newAnswer(Object answer) {
   ETAnswer a = new ETAnswer(answer);
   a.setTutorial(tutorial);
   a.setMsgNotFeedback(features.get("notFeedbackErrorMsg"));
   a.setMsgNotFeedbackTitle(features.get("notFeedbackErrorTitle"));
   return a;
}
```

La clase semántica de <e-Tutor>. Parte III.

```
    public ETAbstractAnswer newDefaultAnswer() {
        ETDefaultAnswer a = new ETDefaultAnswer();
        a.setTutorial(tutorial);
        a.setMsgNotFeedback(features.get("notFeedbackErrorMsg"));
        a.setMsgNotFeedbackTitle(features.get("notFeedbackErrorTitle"));
        return a;
    }

    public ETQuestionPoint addAnswer(ETQuestionPoint qp, ETAbstractAnswer a) {
        qp.addAnswer(a);
        return qp;
    }

    public ETAbstractAnswer addFeedback(ETAbstractAnswer a,
                                        ETTutorialElement e) {
        a.addFeedback(e);
        return a;
    }

    public Color newColor(String c) {
        if (c == null) return null;
        return Color.decode("0x"+c);
    }

}
```

La clase semántica de <e-Tutor>. Parte IV.

Apéndice B: Gramática de XLOP

SpecXLOP ::= {*Regla*}+

Regla ::= *NoTerminal* '::=' { *ElementoSintactico* }* '{' { *Ecuacion* }* '}'

ElementoSintactico ::= *NoTerminal* | **#pcdata** | *Elemento*XML

*Elemento*XML ::= *EtiquetaApertura* { *ElementoSintactico* }* *EtiquetaCierre* |
EtiquetaElmVacio

Ecuacion ::= *ReferenciaAtributo* '=' *ExpresionSemantica*

ExpresionSemantica ::= *Funcion* '(' (*ExpresionSemantica* { , *ExpresionSemantica* }*)? ')' |
ReferenciaAtributo | *ValorLiteral*

ReferenciaAtributo ::= *Atributo* **of** (*NoTerminal* | **#pcdata** | *EtiquetaApertura*)
('(' *NumeroOcurrencia* ')')?

Gramática EBNF de alto nivel del lenguaje de especificación de XLOP.